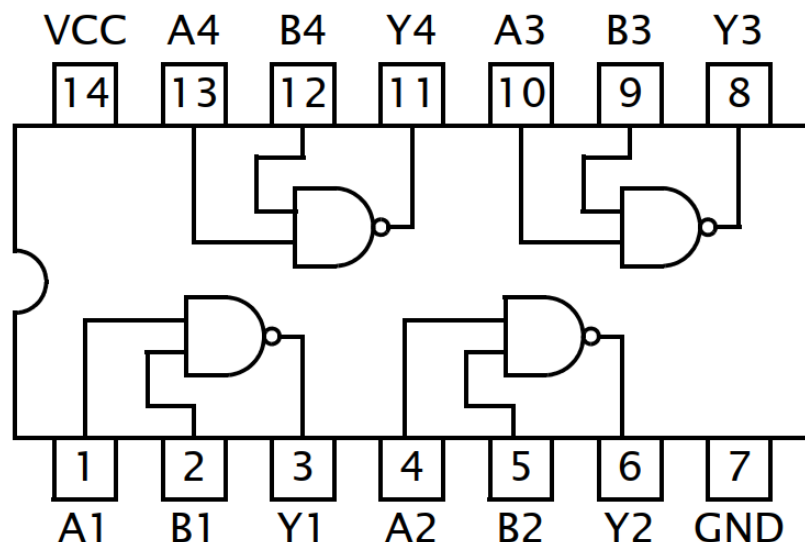


- VHDL – VHSIC Hardware Description Language
 - VHSIC – Very High Speed Integrated Circuit
 - Hardware Description Language (HDL): Hardwarebeschreibungssprache
 - seit 1987 als IEEE–1076–Standard, und wird laufend aktualisiert
 - ursprünglich als Beschreibungs– und Simulationssprache entwickelt
- VHDL eignet sich für Spezifikation, Dokumentation und technologie–unabhängige Beschreibung digitaler Systeme und Schaltungen auf verschiedenen Abstraktionsebenen.
- VHDL ist für Entwürfe komplexer Systeme (FPGA–/CPLD–/ASIC–Design) ausgelegt:
 - getrennte Übersetzung einzelner Module,
 - Top–Down–/Bottom–Up–Entwurf,
 - Modularisierung und Hierarchiebildung (Syntaxkonstrukte wie z.B.: ENTITY, COMPONENT, PROCESS, PROCEDURE usw.).

- Entwurfseinheit (ENTITY)
 - ist ein zusammenhängendes, in sich abgeschlossenes System,
 - ist durch eine Schnittstellenbeschreibung für die Umgebung sichtbar,
 - entspricht einem Symbol in einer grafischen Schaltungsbeschreibung.

- Architektur (ARCHITECTURE)
 - enthält die Beschreibung der Funktionalität der modellierten Komponente.
 - Alle simulierbare/synthäsefähige Entwurfseinheiten haben eine Architekturbeschreibung.
 - Für eine Komponente können mehrere Architekturen existieren:
 - auf unterschiedlichen Abstraktionsebenen (Systemebene, algorithmische Ebene, Registertransferebenen, Logikebene);
 - aus verschiedenen Sichten (Verhalten, Struktur, Geometrie, Test) beschreiben;
 - mit verschiedene Entwurfsalternativen (optimiert für Fläche/Laufzeit (size/speed), zeitsequentiell, fließbandorganisiert).



```
ARCHITECTURE logic OF SN7400 IS
BEGIN
    Y1 <= NOT(A1 AND B1);
    Y2 <= NOT(A2 AND B2);
    Y3 <= NOT(A3 AND B3);
    Y4 <= NOT(A4 AND B4);
END ARCHITECTURE logic;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

-- Quad 2-Input NAND Gate
ENTITY SN7400 IS
    PORT(A1, A2, A3, A4: IN  std_logic;
         B1, B2, B3, B4: IN  std_logic;
         Y1, Y2, Y3, Y4: OUT std_logic);
END ENTITY SN7400;
```

```
ARCHITECTURE structure OF SN7400 IS
BEGIN
    g1: NAND2 PORT MAP(Y1, A1, B1);
    g2: NAND2 PORT MAP(Y2, A2, B3);
    g3: NAND2 PORT MAP(Y3, A3, B3);
    g4: NAND2 PORT MAP(Y4, A4, B4);
END ARCHITECTURE structure;
```

- VHDL ist eine streng (stark) typisierte (typgebunde) Sprache:
 - Objekte eines Typs können nur die durch diesen Typ definierten Werte annehmen, und auf diese Objekte sind nur die für diesen Typ definierten Operationen anwendbar.
 - Automatische oder implizite Typkonvertierungen sind weitgehend ausgeschlossen.
 - Typkonvertierungen müssen explizit mit Konvertierungsfunktionen durchgeführt werden.
 - Bei gleichartigen Typen (numerische Typen oder gleichstrukturierte Reihungstypen) ist eine vereinfachte, sog. Cast-Konvertierung in der Form `Typname(Ausdruck)` vorgesehen.
 - Das strenge Typkonzept erhöht die Zuverlässigkeit:
 - der Compiler kann bereits zur Übersetzungszeit die meisten Semantik-Fehler finden,
 - das Laufzeitsystem kann auf Semantik-Fehler, die zur Laufzeit/Simulation auftreten, gezielt reagieren.

- Vier Datentypklassen in VHDL:
 - skalare Datentypen (scalare types):
 - diskrete Typen:
 - Aufzählungstyp (enumeration type)
 - Bereichstyp (range type)
 - physikalischer Typ (physical type)
 - Gleitkommatyp (real)
 - zusammengesetzte Datentypen (composite types):
 - Reihungstyp (array type)
 - Verbundtyp (record type)

Mit zusammengesetzten Typen lassen sich Daten zu komplexen Datentypen organisieren.
 - Zugriffstyp (access type)
 - Dateityp (file type)
- Vorsicht! Einige Datentypen sind gar nicht synthesefähig, andere nur bedingt.

- Ein Datentyp (TYPE) in VHDL definiert sowohl eine Menge von Werten, die ein Objekt annehmen kann, als auch Operationen, die auf diesen Werten angewendet werden dürfen.
- Es gibt zwei Grundoperationen, die für jeden Datentyp definiert sind: die Zuweisung und der Vergleich auf Gleichheit.
- Eine Untertypdefinition (SUBTYPE) schränkt nur die Wertemenge eines zuvor definierten Datentyps ein, ohne dadurch einen neuen Typ einzuführen.
 - Der Untertyp und der (Basis-)Typ sind kompatibel.
 - Objekte mit verschiedenen Untertypen des gleichen Basistyps können mit den Operatoren des Basistyps verknüpft werden.

- Der Aufzählungstyp definiert eine geordnete Menge von Werten, die durch sog. Aufzählungsliterale (Bezeichner oder Zeichenlitterale) repräsentiert werden. Die Ordnung ist durch die Aufschreibungsreihenfolge vorgegeben.

```
-- vordefinierte Typen aus dem standard-Paket:
TYPE boolean    IS (false, true);
TYPE bit        IS ('0', '1');
TYPE character  IS (.. 256 ASCII-Zeichen ..);

-- selbstdefinierte Aufzählungstypen
TYPE TOPCode    IS (ADD, ADC, SUB, SBC, JMP, MOV, LOAD, STORE);
TYPE TOctValue  IS ('0', '1', '2', '3', '4', '5', '6', '7');
TYPE TState     IS (idle, run, error, send, receive);
```

- Bezeichner oder Zeichenlitterale eines Aufzählungstyps werden implizit mit nicht negativen numerischen Werten von links nach rechts aufsteigend durchnummeriert. Das erste (am weitesten links stehende) Element in der Aufzählung hat die Position Null.

- Der Aufzählungstyp zur Modellierung 9-wertiger Logik

```
-- vordefinierter Typ aus dem std_logic_1164-Paket:

TYPE std_ulogic IS ( 'U',  -- Uninitialized      nicht initialisiert
                    'X',  -- Forcing Unknown    stark unbekannt
                    '0',  -- Forcing 0          starke logische 0
                    '1',  -- Forcing 1          starte logische 1
                    'Z',  -- High Impedance     hochohmig, für Busse
                    'W',  -- weak Unknown       schwach unbekannt
                    'L',  -- weak 0             schwache logische 0
                    'H',  -- weak 1             schwache logische 1
                    '-'   -- Don't care         egal
                );
```


▪ Auswahl häufig benötigter (Konvertierungs-)Funktionen

```
LIBRARY ieee;

USE ieee.std_logic_1164.ALL;
    FUNCTION to_bit (s: std_ulogic; xmap: BIT := '0') RETURN bit;
    FUNCTION to_bitvector (s: std_logic_vector; xmap: bit := '0') RETURN bit_vector;
    FUNCTION to_stdulogic (b: bit) RETURN std_ulogic;
    FUNCTION to_stdlogicvector (b: bit_vector) RETURN std_logic_vector;
    FUNCTION rising_edge (s: std_ulogic) RETURN boolean;
    FUNCTION falling_edge (s: std_ulogic) RETURN boolean;

USE ieee.std_logic_arith.ALL;
    FUNCTION conv_std_logic_vector (ARG: integer; SIZE: integer) RETURN std_logic_vector;
    FUNCTION EXT (ARG: std_logic_vector; SIZE: integer) RETURN std_logic_vector;
    FUNCTION SXT (ARG: std_logic_vector; SIZE: integer) RETURN std_logic_vector;

USE ieee.std_logic_signed.ALL;
    FUNCTION conv_integer (ARG: std_logic_vector) RETURN integer;

USE ieee.std_logic_unsigned.ALL;
    FUNCTION conv_integer (ARG: std_logic_vector) RETURN integer;
```

- Der Bereichstyp (RANGE) legt den Wertebereich des Typs durch eine explizite Angabe einer Ober- und einer Untergrenze fest.
 - Die Angaben zur Ober- und Untergrenze müssen konstante Ausdrücke sein, also zur Übersetzungszeit berechenbar.
 - Mit den Schlüsselwörtern TO und DOWNTO lassen sich aufsteigende und absteigende Wertebereiche vereinbaren.

```
-- vordefinierte Bereichstypen aus dem standard-Paket:
TYPE integer IS RANGE  -2**31 TO 2**31-1;    -- -2147483648 to +2147483647
TYPE real    IS RANGE -1.0E38 TO 1.0E38;     -- -1.0E38 to +1.0E38

-- selbstdefinierte Bereichstypen
TYPE TIndex      IS RANGE 15 DOWNTO 0; -- ganzzahliger Bereichstyp
TYPE TCounter    IS RANGE 0 TO 4095;   -- ganzzahliger Bereichstyp
TYPE TProbability IS RANGE 0.0 TO 1.0;
```

- Ein ganzzahliger Bereichstyp umfasst alle Zahlen zwischen der Ober- und der Untergrenze. Alle Operationen mit Daten dieses Typs sind genau und entsprechen den üblichen arithmetischen Gesetzen.

- Mit einer Untertypdefinition (SUBTYPE) kann man die Wertemenge eines zuvor definierten Bereichs- oder Aufzählungstyps einschränkt, ohne dadurch einen neuen Datentyp einzuführen.
 - Operationen können auf Typen und Untertypen in gleicher Weise angewendet werden, sofern keine Bereichsbeschränkungen verletzt werden.

```
-- vordefinierte Typen aus dem standard-Paket:
SUBTYPE natural IS integer RANGE 0 TO integer'HIGH;
SUBTYPE positive IS integer RANGE 1 TO integer'HIGH;

-- vordefinierte Typen aus dem std_logic_1164-Paket:
SUBTYPE x01z IS std_ulogic RANGE 'X' TO 'Z'; -- ('X','0','1','Z')
SUBTYPE ux01 IS std_ulogic RANGE 'U' TO '1'; -- ('U','X','0','1')

-- selbstdefinierte Typen
TYPE elektronische_elemente IS (spule, kondensator, widerstand, leitung, klemme,
    diode, bipolarer_transistor, unipolarer_transistor);
SUBTYPE passive_elemente IS elektronische_elemente RANGE spule TO klemme;
SUBTYPE aktive_elemente IS elektronische_elemente
    RANGE diode TO unipolarer_transistor;
```

- Der Reihungstyp (ARRAY) ist eine sog. homogene Struktur, die sich aus Komponenten zusammensetzt, die alle vom selben Datentyp sind.
 - Die Definition eines Reihungstyps legt sowohl den Typ der Komponenten als auch den Typ der Indizes fest.
 - Die Anzahl der Indizes definiert die Dimension eines Reihungstyps. Indextypen müssen diskrete Typen sein: (Aufzählungstypen oder Bereichstypen). Indextypen verschiedener Dimensionen eines Reihungstyps können unterschiedlich sein.

```
-- vordefinierte Reihungstypen aus dem std_logic_1164-Paket:
TYPE stdlogic_1d    IS ARRAY (std_ulogic) OF std_ulogic;
TYPE stdlogic_table IS ARRAY (std_ulogic, std_ulogic) OF std_ulogic;

-- selbstdefinierte Typen
TYPE TMonat        IS (JAN, FEB, MAR, APR, MAI, JUN, JUL, AUG, SEP, OKT, NOV, DEZ);
TYPE TTag          IS (MO, DI, MI, DO, FR, SA, SO);
TYPE Twoche        IS RANGE 1 TO 52;

TYPE TKalender IS ARRAY (TTag, Twoche, TMonat) OF positive;
```

- In VHDL unterscheidet man zwischen eingeschränkten und uneingeschränkten (RANGE <>) Reihungstypen.
 - Bei eingeschränkten Reihungstypen sind die Indexbereiche (also die oberen und unteren Grenzen) aller Objekte des Typs gleich. Somit haben alle Objekte eines solchen Typs die gleiche Größe.
 - Bei uneingeschränkten Reihungstypen können verschiedene Reihungsobjekte unterschiedliche Grenzen haben.

```
-- vordefinierte uneingeschränkte Reihungstypen aus dem std_logic_1164-Paket:  
TYPE std_ulogic_vector IS ARRAY (natural RANGE <>) OF std_ulogic;  
TYPE std_logic_vector  IS ARRAY (natural RANGE <>) OF std_logic;  
  
-- vordefinierte uneingeschränkte Reihungstypen aus dem Package standard  
TYPE string            IS ARRAY (positive RANGE <>) OF character;  
TYPE bit_vector        IS ARRAY (natural RANGE <>) OF bit;  
  
-- vordefinierte uneingeschränkte Reihungstypen aus dem Package std_logic_arith  
TYPE unsigned          IS ARRAY (natural RANGE <>) OF std_logic;  
TYPE signed            IS ARRAY (natural RANGE <>) OF std_logic;
```

- Konstanten sind Objekte mit festen Werten, die während der Ausführung eines Modells nicht geändert werden können.
- Die Deklaration einer Konstante (CONSTANT) gibt den Namen der Konstante, ihren Datentyp und optional ihren Initialisierungswert an.
- In VHDL werden sog. offene Konstanten unterstützt, die nur in Paketen deklariert werden dürfen und deren Wert erst im Pakettrumpf spezifiziert wird.

```
CONSTANT tcyc      : time := 20 ns;  
  
CONSTANT delimiter : bit_vector(0 TO 7) := "01111110";  
  
CONSTANT message   : string := "Segmentation fault";  
  
CONSTANT N         : natural := 16;  
CONSTANT muster    : std_logic_vector(N-1 DOWNT0 0) := x"A55A";
```

- Variablen sind Objekte mit veränderbaren Werten
 - Sie dienen zur lokalen Aufbewahrung von temporären Daten und werden hauptsächlich in sequentiellen Bereichen (Unterprogrammen oder Prozessen) eingesetzt.
 - Das Verhalten von Variablen in VHDL entspricht weitgehend dem von Variablen aus bekannten Programmiersprachen: sie nehmen ihren neuen Wert unmittelbar nach der Zuweisung (:=) an.
 - In VHDL'93 werden sog. gemeinsame Variablen (shared variables) unterstützt. Sie werden im Deklarationsbereich einer Architektur vereinbart, und auf sie darf in Prozessen lesend und schreibend zugegriffen werden (nicht deterministisch, nicht synthetisierbar).

```
VARIABLE counter : TCounter := 0;           -- explizite Initialisierung mit 0
VARIABLE delta   : real      := 0.2;         -- explizite Initialisierung mit 0.2

VARIABLE tmp      : positive;                -- implizite Initialisierung mit 1
VARIABLE adresse  : integer RANGE 0 TO 16#FFFF#; -- implizite Initialisierung mit 0
```

- Signale sind Objekte mit veränderbaren Werten und einem Zeitverhalten
 - Signale haben ein anderes Verhalten als Variablen: sie bekommen einen neuen Wert nicht sofort sondern erst nach einer gewissen Zeit zugewiesen, z.B. am Ende eines Prozesses oder nach Ablauf einer vorgegebenen Verzögerung.
 - Signale dienen zur Verbindung einzelner Komponenten innerhalb eines Entwurfes oder zur Modellierung zeitlichen Verhaltens digitaler Systeme.
 - Initialisierung von Signalen (auch Variablen) bei deren Deklaration wird während der Synthese komplett ignoriert.

```
SIGNAL cnt    : TCounter    := 0;           -- Initialisierung vermeiden!  
SIGNAL dff    : std_logic   := '0';         -- Initialisierung vermeiden!  
SIGNAL data   : std_logic_vector(0 TO 7) := "00000000"; -- Initialisierung vermeiden!  
SIGNAL acc    : integer RANGE -2047 TO 2047;
```


- Die Schnittstellenbeschreibung spezifiziert den Namen einer Entwurfseinheit und gibt deren Schnittstellen an.
- Die Schnittstellenbeschreibung umfasst vier optionale Definitionsbereiche mit
 - Interfacekonstanten (generic constants),
 - Interfacesignalen (ports),
 - einen Deklarationsbereich mit weiteren Vereinbarungen für gemeinsame Konstanten, Typen, Unterprogramme usw., aber keine Variablen,
 - einem Anweisungsbereich (nicht synthesefähig).

```
ENTITY myUnit IS
  GENERIC ( . . . ); -- Interfacekonstanten in einer generische Konstanten
  PORT ( . . . );    -- Interfacesignale in einer Portdeklaration
  -- optionaler Deklarationsbereich
  -- optionaler Anweisungsbereich
END myUnit;
```

- Interfacekonstanten (GENERIC) stellen eine einheitliche Schnittstelle zur Parametrisierung einer Entwurfseinheit dar.
 - Innerhalb einer Entwurfseinheit verhalten sich Interfacekonstanten wie gewöhnliche Konstanten.
 - Außerhalb einer Entwurfseinheit können Interfacekonstanten mit neuen Werten bei der Instanziierung einer Komponente belegt werden.

```
ENTITY UniComCon IS
  GENERIC(LENDEF: natural := 32;
          RSTDEF: std_logic := '0';
          MSBDEF: boolean := true);
  . . .
END UniComCon;
```



```
u1: UniComCon
  GENERIC MAP(LENDEF => 16,
              RSTDEF => RSTDEF,
              MSBDEF => true);
```



```
u2: UniComCon
  GENERIC MAP(LENDEF => 8,
              RSTDEF => '1',
              MSBDEF => false);
```

- Interfacesignale (PORT) bilden die eigentlichen Kommunikations-schnittstellen zwischen einer Entwurfseinheit und der Einsatzumgebung durch Angaben von Namen, Typen und Signalflussrichtungen.

Modus	Eigenschaft	Kommentar
IN	unidirektionaler Eingangsport	Der Datenfluss geht nur in das System hinein.
OUT	unidirektionaler Ausgangsport	Der Datenfluss geht nur aus dem System heraus.
INOUT	bidirektionaler Port	Der Datenfluss geht in beide Richtungen und kann mehrere Treiber haben. Standardeinstellung, wenn kein Modus angegeben ist.
BUFFER	bidirektionaler Port	Der Datenfluss geht in beide Richtungen, ein Port darf nur einen Treiber haben.


```
-- UART und I2C
ENTITY UniComCon IS
    PORT(TXD: OUT    std_logic;
          RXD: IN     std_logic;
          SCL: OUT    std_logic;
          SDA: INOUT  std_logic);
END UniComCon;
```

```
u1: UniComCon
PORT MAP(TXD => TXD1,
          RXD => RXD1,
          SCL => SCL1,
          SDA => SDA1);
```

```
u2: UniComCon
PORT MAP(TXD => TXD2,
          RXD => RXD2,
          SCL => OPEN,
          SDA => OPEN);
```

- Die Architekturbeschreibung (ARCHITECTURE) spezifiziert Beziehungen zwischen Ein- und Ausgängen einer Entwurfseinheit und bestimmt die Struktur, den Datenfluss oder das Verhalten der Entwurfseinheit.
 - Im Unterschied zu herkömmlichen, rein sequentiellen Programmiersprachen, werden alle Anweisungen innerhalb des Anweisungsbereiches einer Architektur parallel ausgeführt. Die Reihenfolge, in der solche Anweisungen innerhalb des Anweisungsbereiches einer Architektur notiert sind, ist ohne Bedeutung.
 - Parallele Anweisungen dienen zur strukturalen Modellierung oder zur Verhaltensmodellierung digitaler Systeme

```
ARCHITECTURE myArch OF myUnit IS
  -- Deklarationsbereich
BEGIN
  -- paralleler Anweisungsbereich
END myArch;
```



- einfache Signalzuweisung
- bedingte Signalzuweisung
- selektierte Signalzuweisung
- Prozessanweisung
- Instanziierung von Komponenten
- Generierungsanweisung
- Blockanweisung
- Prozeduraufruf

- Einfache Signalzuweisungen gehören zu den wichtigsten Elementaranweisungen in VHDL. Sie haben eigenen Zuweisungsoperator (\leq).
- Sie dient dazu, einem Signal-Objekt eine Wellenform bestehend aus Werte- und Zeitstempel-Paaren zuzuweisen.
- Es kommen verschiedene Verzögerungsmodellen (TRANSPORT, AFTER, INERTIAL) zum Einsatz, aber nur „verzögerungsfreie“ Signalzuweisungen sind synthesefähig.

```
CONSTANT RSTDEF: std_logic := '1';
CONSTANT tcyc:  time      := 10 ns;

SIGNAL rst: std_logic := RSTDEF;
SIGNAL clk: std_logic := '0';

-- nicht synthesefähige Signalzuweisungen
rst <= RSTDEF, NOT RSTDEF AFTER 100 ns;
clk <= NOT clk AFTER tcyc/2;
```

```
SIGNAL SCL: std_logic;
SIGNAL TXD: std_logic;
SIGNAL DFF1: std_logic;

-- synthesefähige Signalzuweisungen
SCL <= '0';
TXD <= '1';
DFF1 <= RXD;
. . .
```

- Bedingte Signalzuweisungen basieren auf mehreren Zuweisungsalternativen, die jeweils durch Bedingungen gesteuert werden.
 - Das Verhalten einer bedingten Signalzuweisung ähnelt dem einer sequentiellen IF-ELSE-Anweisung, und entspricht technisch einem priorisierten, verketteten Decoder.
 - In den einzelnen Bedingungen können unterschiedliche Signale oder Signalkombinationen abgefragt werden.

```
CONSTANT tcyc:    time := 10 ns;

SIGNAL hlt: boolean: := false;
SIGNAL clk: std_logic := '0';

-- nicht synthesefähige Signalzuweisungen
-- Clock-Generator mit einem Halte-Signal
clk <= NOT clk AFTER tcyc/2 WHEN NOT hlt ELSE '0';
. . .
hlt <= true;
```

```
CONSTANT N: natural := 16;

SIGNAL reg:  std_logic_vector(N-1 DOWNT0 0);
SIGNAL dbus: std_logic_vector(N-1 DOWNT0 0);
SIGNAL OE:   std_logic; -- output enable

-- Tristate-Bustreiber
dbus <= reg WHEN OE='1' ELSE (OTHERS => 'Z');
```

- Selektierte Signalzuweisungen basieren auf einer Auswahl aus einer Reihe von gleichberechtigten Alternativen.
 - Das Verhalten einer selektierten Signalzuweisung ähnelt dem einer sequentiellen CASE-Anweisung, und entspricht technisch einem Multiplexer.
 - Die selektierte Signalzuweisung wird von einem Ausdruck mit diskretem Wertebereich gesteuert.

```
SIGNAL sel: std_logic_vector(1 TO 2);  
SIGNAL tmp: std_logic_vector(1 TO 3);  
  
WITH sel SELECT  
tmp <= "000" WHEN "00",  
      "011" WHEN "01",  
      "101" WHEN "10",  
      "110" WHEN "11",  
      "---" WHEN OTHERS;
```

- Design-Regeln zur Modellierung von Schaltnetzen
 - einfache Schaltnetze (z.B.: Multiplexer, Decoder, Basiszellen von Schaltketten)
 - Digitaltechnik: Beschreibung durch Funktionstabellen
 - VHDL: Nachbildung der tabellarischen Beschreibung mit bedingten oder selektierten Signalzuweisungen; effizient, kompakt, übersichtlich, eindeutig, leicht modifizierbar, technologieunabhängig
 - algorithmische (Verhaltens-)Beschreibung mit Prozessen und sequentiellen Anweisungen kann manchmal mehrdeutig, unübersichtlich sein.
 - Beschreibung mit booleschen Gleichungen ist meistens unnötig, und stellt bereits das Ergebnis einer Synthese oder Minimierung dar.
 - zusammengesetzte Schaltnetze (Schaltketten, z.B.: arithmetische Schaltketten wie N-Bit-Addierer, N-Bit-Vergleicher)
 - Beschreibung in der Digitaltechnik durch hierarchische Strukturen
 - Beschreibung in VHDL als strukturelle Funktionsbeschreibungen mit Hilfe generischer Komponenten, meistens wird die Vektorlänge als skalierbare Größe gewählt.

▪ 1-aus-4-Multiplexer mit Enable

en	sel	y
0	--	0
1	00	x(0)
1	01	x(1)
1	10	x(2)
1	11	x(3)

```
SIGNAL x:  std_logic_vector(0 TO 3);  
SIGNAL y:  std_logic;  
SIGNAL t:  std_logic;  
SIGNAL en: std_logic;
```

```
SIGNAL sel: std_logic_vector(1 TO 2);  
    . . .  
WITH sel SELECT  
    t <= x(0) WHEN "00",  
        x(1) WHEN "01",  
        x(2) WHEN "10",  
        x(3) WHEN OTHERS;  
  
y <= t WHEN en='1' ELSE '0';
```

```
TYPE TState IS (S0, S1, S2, S3);  
SIGNAL state: TState;  
    . . .  
WITH state SELECT  
    t <= x(0) WHEN S0,  
        x(1) WHEN S1,  
        x(2) WHEN S2,  
        x(3) WHEN S3;  
  
y <= t WHEN en='1' ELSE '0';
```

```
SIGNAL adr: integer RANGE 0 TO 3;  
    . . .  
WITH adr SELECT  
    t <= x(0) WHEN 0,  
        x(1) WHEN 1,  
        x(2) WHEN 2,  
        x(3) WHEN 3;  
  
y <= t WHEN en='1' ELSE '0';
```

▪ 1-aus-4-Dekoder mit Enable

en	sel	y
0	--	0000
1	00	1000
1	01	0100
1	10	0010
1	11	0001

```
SIGNAL y: std_logic_vector(0 TO 3);
SIGNAL t: std_logic_vector(0 TO 3);
SIGNAL en: std_logic;
```

```
SIGNAL sel: std_logic_vector(1 TO 2);
    . . .
WITH sel SELECT
    t <= "1000" WHEN "00",
        "0100" WHEN "01",
        "0010" WHEN "10",
        "0001" WHEN OTHERS;

y <= t WHEN en= '1'
    ELSE (OTHERS => '0');
```

```
TYPE TState IS (S0, S1, S2, S3);
SIGNAL state: TState;
    . . .
WITH state SELECT
    t <= "1000" WHEN S0,
        "0100" WHEN S1,
        "0010" WHEN S2,
        "0001" WHEN S3;

y <= t WHEN en='1'
    ELSE (OTHERS => '0');
```

```
SIGNAL adr: integer RANGE 0 TO 3;
    . . .
WITH adr SELECT
    t <= "1000" WHEN 0,
        "0100" WHEN 1,
        "0010" WHEN 2,
        "0001" WHEN 3;

y <= t WHEN en='1'
    ELSE (OTHERS => '0');
```