Datentypen

Type Declaration

- **type** *identifier* **is** *type_definition ;*
- *Beispiele:*

```
type apples is range 0 to 100;
type vec is std_logic_vector (17 downto 3);
```


Integer Types

- `signal s : integer;`
- *Bereich ist nicht genau definiert (implementierungsabhängig), reicht aber auf jeden Fall von -2,147,483,647 bis +2,147,483,647 ($-2^{31}+1$ bis $+2^{31}-1$).*

Integer Types (cont.)

- **signal** *s :* **range** <*expr*> **to** <*expr*> *;*
- **signal** *s :* **range** <*expr*> **downto** <*expr*> *;*
- <*expr*> *muss Integer sein*
- **to** *: aufsteigender Bereich (ascending range)*
- **downto** *: absteigender Bereich (descending range)*
- *Defaultwert: »linksester« Wert*

Integer Constants

- *Dezimalzahlen: 23, 0, 146*
- *Unterstriche erhöhen die Lesbarkeit: 7_000_000*
- *Exponential Notation: 46E5, 1E+12, 19e00*
- *Based Literals: Basis ist dezimal gegeben!*
  - *2#1111_1101# = 16#FD# = 16#0fd# = 253*
  - *2#1#E10 = 16#4#E2 = 10#1024#E+00*

Operationen auf Integern

- `+` *Addition*
- `-` *Subtraktion oder Negation*
- `*` *Multiplikation*
- `/` *Division*
- `mod` *Modulo*
- `rem` *Remainder*
- `abs` *Absolutbetrag*
- `**` *Exponentiation*

Division / Remainder

- *Division A / B*
  - *Ergebnis ist Division von A durch B*
  - *Rundung in Richtung 0*
  - *Es gilt: (-A) / B = -(A / B) = A / (-B)*
- *Remainder*
  - *Definiert durch A = (A / B) * B + (A* **rem** *B)*
  - *Hat das gleiche Vorzeichen wie A*
  - *Absolutwert ist kleiner als der von B*

Beispiele

- *5* **rem** *3 = 2*
- *(-5)* **rem** *3 = -2*
- *5* **rem** *(-3) = 2*
- *(-5)* **rem** *(-3) = -2*

Die Klammern sind von der Grammatik von VHDL vorgeschrieben.

Modulo

- *Definiert durch*
  *A = B * N + (A* **mod** *B); -- für N integer*
- *Gleiches Vorzeichen wie B*
- *Absolutwert kleiner als der von B*
- *Beispiele:*
  - *5* **mod** *3 = 2; (-5)* **mod** *3 = 1;*
  - *5* **mod** *(-3) = -1; (-5)* **mod** *(-3) = -2;*

Exkurs: Synopsys

- *Im Synopsys Design Compiler ist die Division nur durch konstante Zweierpotenzen erlaubt*
- *Die Division wird durch einen Shift berechnet*
- *Problem: bei negativen Zahlen entspricht das Ergebnis nicht dem Standard*

Floating-Point Types

- *Darstellung von reellen Zahlen*
- *Vordefiniert: Datentyp real*
  - *reicht mindestens von -1.0E+38 bis +1.0E+38,*
  - *hat mindestens 6 Dezimalstellen Genauigkeit,*
  - *entspricht dem IEEE 32-Bit Standard*
- **Sub-Typen:**

```
type inp_level is range -10.0 to +10.0;
```


Floating-Point Types (cont.)

- *Operationen:*
  - +, -, *, /, **abs**, **
  - *Argumente müssen gleichen Typ haben, aber*
  - *bei ** muss rechtes Argument Integer sein*
- *Defaultwert: »linksester« Wert*
- *Fließkommatypen sind in der Regel nicht synthetisierbar*

Physical Types

- *Stellen physikalische Einheiten wie Länge, Masse, Zeit, Spannung dar.*
- *Die Definition beinhaltet eine Grundeinheit.*
- *Beispiel:*

```
type resistance is range 0 to 1E9
    units
       ohm;
    end units;

signal s : resistance := 5 ohm;
```


Physical Types (cont.)

```
type resistance is range 0 to 1E9
   units
      ohm;
      kohm = 1000 ohm;
      Mohm = 1000 kohm;
   end units;

type length is range 0 to 1E9
   units
      um;               -- primary unit: micron
      mm   = 1000 um;   -- metric units
      m    = 1000 mm;
      mil  =  254 um;   -- imperial units
      inch = 1000 mil;
   end units;
```


Physical Type »time«

- **time** *ist vordefiniert in VHDL*

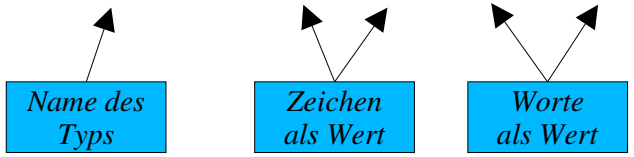
```
type time is range implementation_defined
   units
      fs;
      ps = 1000 fs;
      ns = 1000 ps;
      us = 1000 ns;
      ms = 1000 us;
      sec = 1000 ms;
      min = 60 sec;
      hr  = 60 min;
   end units;
```

- *Kann zur Angabe von Verzögerungszeiten verwendet werden*

Enumeration Types

- *Werte können einen Namen erhalten*
- *Bezeichner können identifier oder character sein*
- *Beispiel:*

```
type opcode is ( '+', '-', swap, idle );
```



Enumeration Types (cont.)

- *Können in Zuweisungen verwendet werden*

```
signal s : opcode;
s <= idle;
```

- *Vordefinierte Operationen:*

  `=, /=, <, <=, >, >=.`

  Die Position in der Aufzählung wird verglichen

```
type alu_op is (nop, add, sub, mul);
-- es gilt
add < sub
nop /= add
mul >= sub
```


Vordefinierte Enumeration Typen

- `type severity_level is ( note, warning, error, failure );`

  für assertions.

- `type character is`

```
( nul, soh, stx, etx, eot, enq, ack, bel,
  bs,  ht,  if,  vt,  ff,  cr,  so,  si,
  ..., 'y', 'z', '{', '|', '}', '~', del );
```

  VHDL-87 verwendet ASCII (128 Zeichen).

- *VHDL-93 erlaubt alle 256 Zeichen.* VHDL93

Vordefinierte Enumeration Typen: Boolean

- `type boolean is ( false, true );`
- *Repräsentiert das Ergebnis von relationalen und logischen Operatoren*
- `123 = 123, 'a' = 'a', 7 ns = 7 ns,`
  `123 < 456, 'a' < 'b', 3 ps < 3 ps` *sind* `true`
- `123 = 456, 'a' = 'A', 7 ns = 2 us,`
  `123 > 456, 'a' < 'a', 7 us < 7 ns` *sind* `false`.
- *Die Operatoren* **`and, or, nand, nor, xor, not`** *sind auf boolean definiert:*

```
(b /= 0) and (a / b > 1)
```


Vordefinierte Enumeration Typen: Bit

- `type bit is ( '0', '1' );`
- *'0' und '1' sind überladen, da sie auch Teil von character sind.*
- *Modellieren logische Werte in Hardware*
- *Die Operatoren* **`and, or, nand, nor, xor, not`** *sind auch auf Bit definiert:*

```
'0' and '1' = '0', '1' xor '1' = '0'
```

- *Können nicht mit boolean verknüpft werden*

  `'0' and true` *ist illegal.*

Enumeration und Synthese

- *Enumeration Typen sollten zur Modellierung von Objekten verwendet werden, die nur **diskrete Zustände** annehmen können (z.B. Zustände von Automaten)*
- *Der erste Wert eines Enumeration Typs sollte dem Default/Anfangs-Wert entsprechen*
- *Die Verwendung von gleichen Elementnamen in verschiedenen Enumeration Typen sollte vermieden werden*

Subtypen

- ```
  subtype klein is integer range -128 to 127;
  signal deviation : klein;
  signal adjustment : integer;
  deviation <= deviation + adjustment;
  ```
- *Definiert einen Typ, der den Basistyp einschränkt*
- *Es ist ein Fehler, wenn der Wert nicht im Bereich dargestellt werden kann.*
- *Vordefiniert ist*
  ```
  subtype natural is integer range 0 to maxint;
  subtype positive is integer range 1 to maxint;
  ```


Subtypen (cont.)

- ```
  type logic_level is
              (unknown, low, undriven, high);

  subtype valid_level is logic_level
              range low to high;
  ```
- *Schließt alle Elemente dazwischen mit ein*
- *Unterscheidung ist möglich mittels*
  `logic_level'(high), valid_level'(high)`
  (Type Qualification)

Subtypen und Synthese

- *Subtypen sollten verwendet werden, um Bereichsfehler früh erkennen zu können*
- *Die abgeleiteten Typen bleiben kompatibel*
  ```
  signal s    : integer;
  signal t, u : integer range 2 to 7;
  t <= s + u;
  ```
- *Die Größe von Registern / Ports kann dadurch bestimmt werden*
  ```
  signal reg : integer range 0 to 15; -- 4 Bit
  ```

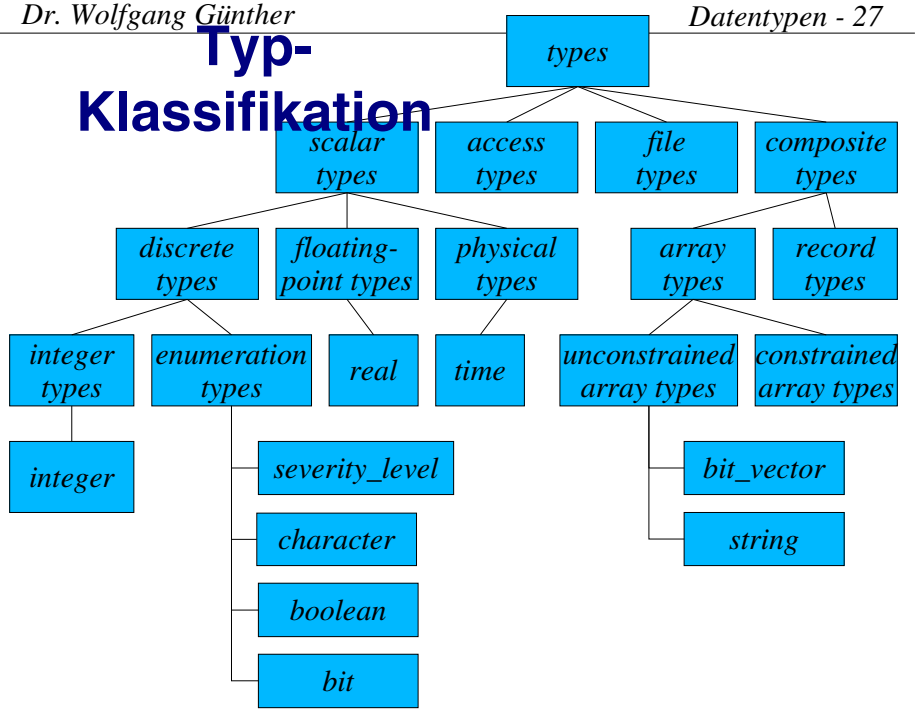
Attribute auf Scalar Types

- *T sei ein Scalar type*
  - *T'left* — *erster (linker) Wert von T*
  - *T'right* — *letzter (rechter) Wert von T*
  - *T'low* — *kleinster Wert von T*
  - *T'high* — *größter Wert von T*
  - *T'ascending* — *true, wenn T aufsteigend ist*
  - *T'image(x)* — *ein String, der den Wert von x darstellt*
  - *T'value(s)* — *der Wert in T, durch s dargestellt*

Beispiele

```
type foo is integer range 21 downto 11;
foo'left = 21;
foo'right = 11;
foo'low = 11;
foo'high = 21;
foo'ascending = false;
foo'image(14) = Ž"14";
foo'value("20") = 20;
```


Typ-Klassifikation



Zusammengesetzte Typen

- *Records*
  - *Gruppe von Elementen unterschiedlichen Typs*
  - *Elemente werden durch ihren Namen adressiert*
  - *äquivalent zu »struct« in C*
- *Arrays*
  - *Gruppe von Elementen gleichen Typs*
  - *Elemente werden über einen Index adressiert*

Records

- ```
  type channel is record
                    data : integer;
                    address : integer;
                    direction : bit;
                  end record;
  ```
- *Zugriff auf einzelne Elemente:*
  ```
  signal line1 : channel;
  signal b : bit;
  b <= line1.direction;
  ```
- *Record aggregates:*
  ```
  line1 <= ( 7, 15, '0' );
  ```


Record Aggregates

- *Ein Record Aggregate legt alle Elemente fest.*
- *positional association:*
  ```
  line1 <= ( 7, 15, '0' );
  ```
- *named association:*
  ```
  line1 <= ( direction => '0',
             address => 7, data => 15 );
  ```
- *Elemente können mit '|' verknüpft werden,* **others** *bezeichnet alle noch nicht angegebenen:*
  ```
  line1 <= ( address | data => 15,
             others => '0' );
  ```


Arrays

- *Es gibt zwei Arten von Arrays*
  - *Constrained Arrays: die Array-Grenzen liegen fest*
  - *Unconstrained Arrays: weder Array-Grenzen noch Array-Größe liegen fest*
- *Synthetisierbar sind nur Arrays mit festen Grenzen. Deshalb muss die tatsächliche Größe bei der Erzeugung festgelegt sein*

Constrained Arrays

- ```
  type word is array (0 to 31) of bit;
  ```

- *Der Wertebereich muss diskret sein*
- *Die Indexlaufrichtung kann steigend oder fallend sein*

Beispiele

- ```
  type word is array(31 downto 0) of bit;
  ```
- ```
  type controller_state is
            (initial, idle, active, error);
  type state_counts is array (idle to error)
            of natural;
  ```
  oder
  ```
  type state_counts is array (controller_state
            range idle to error) of natural;
  ```
- ```
  signal s, t : word;
  s(0) <= '0';
  t <= s;
  ```


Mehrdimensionale Arrays

- ```
  type symbol is ('a', 't', 'd', digit, cr);
  type state is range 0 to 6;
  type trans_matrix is array
                    (state, symbol) of state;

  signal s : trans_matrix;
  s(5, 'd') <= 6;
  ```
- *Nicht alle Synthese-Tools unterstützen Arrays mit vielen Dimensionen!*

Array Aggregates

- *positional association:*
  ```
  type point is array (1 to 3) of integer;
  constant origin : point := (0, 0, 0);
  ```
- *named association:*
  ```
  constant coeff : point
                := (1 | 2 => 1, others => 3);
  ```
  oder
  ```
  constant coeff : point
                := (1 to 2 => 1, others => 3);
  ```
- *»others« steht wieder für alle nicht angegebenen Elemente*

Array Attributes

- *A: ein Arraytyp oder Arrayobjekt*
- *A'left* — *Linke Grenze des Indexbereichs*
- *A'right* — *Rechte Grenze des Indexbereichs*
- *A'low* — *Kleinste Grenze des Indexbereichs*
- *A'high* — *Größte Grenze des Indexbereichs*
- *A'range* — *Indexbereich der Dimension N*
- *A'reverse_range* — *... in umgedrehter Richtung*
- *A'length* — *Länge des Indexbereichs*
- *A'ascending* — *true, wenn aufsteigend sortiert ist*

VHDL93

Array Attributes (cont.)

- *Wenn das Array mehrere Dimensionen hat, kann auf die anderen Dimensionen mittels*
  `A'left(2)`, ...
  zugegriffen werden.
- *Beispiel:*
```
type point is array (1 to 3) of integer;
type pos is array (point'range) of real;
```


Unconstrained Arrays

- ```
  type sample is array (natural range <>)
                       of integer;
  ```
- *<> wird auch »box« genannt*
- *Um ein Objekt zu deklarieren, muß der Bereich festgelegt werden:*
  ```
  signal s : sample(0 to 63);
  ```
  oder:
  ```
  subtype long_sample is sample(0 to 255);
  signal t : long_sample;
  ```
  oder bei Konstanten:
  ```
  constant u : sample := ( 1 => 23, 2 => 7 );
  ```


Vordefinierte Arrays

- *Strings:*
  ```
  type string is array (positive range <>)
                       of character;
  ```
- **Bitvektoren**:
  ```
  type bit_vector is array (natural range <>)
                       of bit;
  ```
- *Beispiel:*
  ```
  subtype byte is bit_vector(7 downto 0);
  signal s, t : bit_vector(1 to 4);
  t <= not s;
  ```


Standard-Logic Arrays

- *Im Package ieee.std_logic_1164 ist definiert:*
  ```
  type std_logic_vector is array
              (natural range <>) of std_logic;
  ```
- *Dort sind auch Operationen darauf definiert:*
  ```
  function "and" (l, r : std_logic_vector)
              return std_logic_vector;
  ```
  ebenso
  nand, or, nor, xor, xnor, nor
  ... und viele weitere
  ```

String und Bit-String Literals

- `constant message : string := "Ready  ";`
- `constant c : std_logic_vector(0 to 5) := "ZZ--XX";`
- *Binär:*
  `constant d : bit_vector(0 to 3) := B"0011";`
- *Oktal (Basis 8):*
  `constant e : bit_vector(0 to 8) := O"777";`
- *Hexadezimal (Basis 16):*
  `constant f : bit_vector(0 to 15) := X"FFFF";`

Array Operations

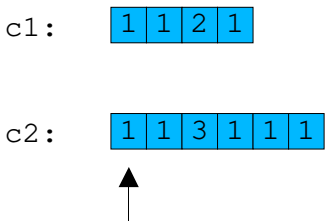
- *Logische Operatoren* **and**, **nand**, **or**, **nor**, **xor**
- **Concatenation Operator &:**
  `b"01" & b"101" = b"01101";`
  `"abc" & 'd' = "abcd";`
  `'a' & 'b' = "ab".`
- **Relationale Operatoren =, /=, <, <=, >, >=**
  - **Operanden brauchen nicht gleich lang zu sein**
  - **Bei = und /= ist der Basistyp beliebig, sonst skalar oder diskret**
  - **Ergebnis: boolean**

Vergleich von Vektoren

- *Vergleich a < b ergibt:*
  - *Wenn a und b Länge 0 haben, ist a < b = false.*
  - *Wenn a Länge 0 hat, b Länge > 0, ist a < b = true.*
  - *Wenn a Länge > 0 hat, b Länge 0, ist a < b = false*
  - *Wenn a und b Länge > 0 haben*
    - *Wenn a(1) < b(1), dann ist a < b = true*
    - *Wenn a(1) > b(1), dann ist a < b = false*
    - *Sonst: Ergebnis ist (Rest von a) < (Rest von b).*

Beispiel

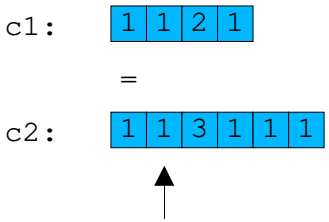
- ```
  type t1 is array(0 to      3) of integer;
  type t2 is array(2 downto -3) of integer;
  constant c1 : t1 := (2 => 2, others => 1);
  constant c2 : t2 := (0 => 3, others => 1);
  ```
- *c1 < c2 ?*


  ```

Beispiel

```
type t1 is array(0 to      3) of integer;
type t2 is array(2 downto -3) of integer;
constant c1 : t1 := (2 => 2, others => 1);
constant c2 : t2 := (0 => 3, others => 1);
```

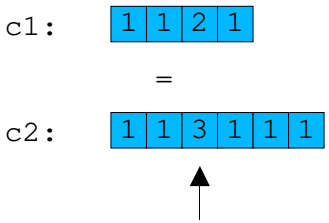
- *c1 < c2 ?*



Beispiel

```
type t1 is array(0 to      3) of integer;
type t2 is array(2 downto -3) of integer;
constant c1 : t1 := (2 => 2, others => 1);
constant c2 : t2 := (0 => 3, others => 1);
```

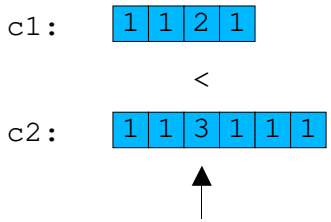
- *c1 < c2 ?*



Beispiel

```
type t1 is array(0 to      3) of integer;
type t2 is array(2 downto -3) of integer;
constant c1 : t1 := (2 => 2, others => 1);
constant c2 : t2 := (0 => 3, others => 1);
```

- *c1 < c2 !*



Array Operations VHDL93

- *Logischer Operator* **xnor**
- *Shift-Operatoren* **sll**, **srl**, **sla**, **sra**, **rol**, **ror**

op		B"10001011" op 3
sll	**shift-left logical**	B"01011000"
srl	**shift-right logical**	B"00010001"
sla	**shift-left arithmetic**	B"01011111"
sra	**shift-right arithmetic**	B"11110001"
rol	**rotate-left**	B"01011100"
ror	**rotate-right**	B"01110001"

Array Slices

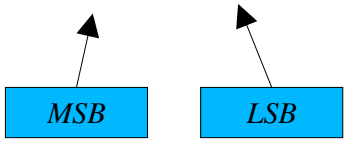
- ```
  type word is std_logic_vector(0 to 15);
  signal s, t : word;
  t(8 to 15) <= s(0 to  7);
  t(0 to  7) <= s(8 to 15);
  ```
- *Beschreiben einen zusammenhängenden Teil*
- *Die Richtung der Definition und des Slices müssen gleich sein, d.h.* `t(7 downto 0)` *ist illegal*
- *Ist der Bereich leer, so spricht man von null slices*
  `t(8 to 0)`

Arrays und Synthese

- *Unconstrained Arrays sollten* **so oft wie möglich** *verwendet werden*
- **Attribute** *'left, 'range, etc.* **verwenden**, *um Feldgrenzen abzufragen*
- *Das linke Bit sollte immer das Most Significant Bit (***MSB***) sein*
- *Grenzen von* **Konstanten** *sollten immer angegeben werden (Lesbarkeit!)*

Arrays und Synthese (2)

- *Die Richtung von Arrays sollte in einem Design immer gleich sein*
- *Beispiele:*
  - *PowerPC: 0* **to** *31*
  - *68000: 31* **downto** *0*



Array Type Conversions

- *Arrays können in einen anderen Typ konvertiert werden, wenn*
  - *der Basistyp gleich ist,*
  - *die Zahl der Dimensionen gleich ist,*
  - *die Indexbereiche müssen »kompatibel« sein*
- ```
  subtype name is string(1 to 20);
  type dstring is array(integer range 0 to 19)
                  of character;
  signal s : name;
  signal t : dstring;
  t <= dstring(s);
  ```

Array Type Conversion (cont.)

- **Keine** *Typkonvertierung ist notwendig, wenn beide Arrays vom gleichen Typ abgeleitet sind und nur verschiedene Grenzen haben*
- ```
  subtype t1 is bit_vector(0 to 15);
  subtype t2 is bit_vector(31 downto 16);
  signal s1 : t1;
  signal s2 : t2;
  s2 <= s1;
  ```


Unconstrained Array Ports

- ```
  entity swap is
     port ( inp  : in  bit_vector;
            outp : out bit_vector);
  end swap;
  ```
- ```
  architecture rtl of swap is
     constant len : integer := inp'length;
     signal help : bit_vector(0 to len-1);
  begin
     help <= inp;
     outp <= help(len/2 to len-1) &
             help(0 to len/2-1);
  end rtl;
  ```
- *Konstanten und Signale brauchen feste Grenzen!*

Multiplexer

- ```
  entity MUX21 is
     port(sel, d0, d1 : in  std_logic;
          dout        : out std_logic);
  end MUX21;
  ```
- ```
  architecture rtl of MUX21 is
  signal s1, s2 : std_logic;
  begin
     s1   <= not sel and d0;
     s2   <=     sel and d1;
     dout <= s1 or s2;
  end rtl;
  ```


Multiplexer (cont.)

- ```
  entity MUX21 is
     port(sel    : in  std_logic;
          d0, d1 : in  std_logic_vector;
          dout   : out std_logic_vector);
  end MUX21;
  ```
- ```
  architecture rtl of MUX21 is
  signal s1, s2 : std_logic_vector(d0'range);
  begin
     assert(d0'range = d1'range and
            d0'range = dout'range)
        report "range mismatch";
     s1   <= not sel and d0;
     s2   <=     sel and d1;
     dout <= s1 or s2;
  end rtl;
  ```
  ```