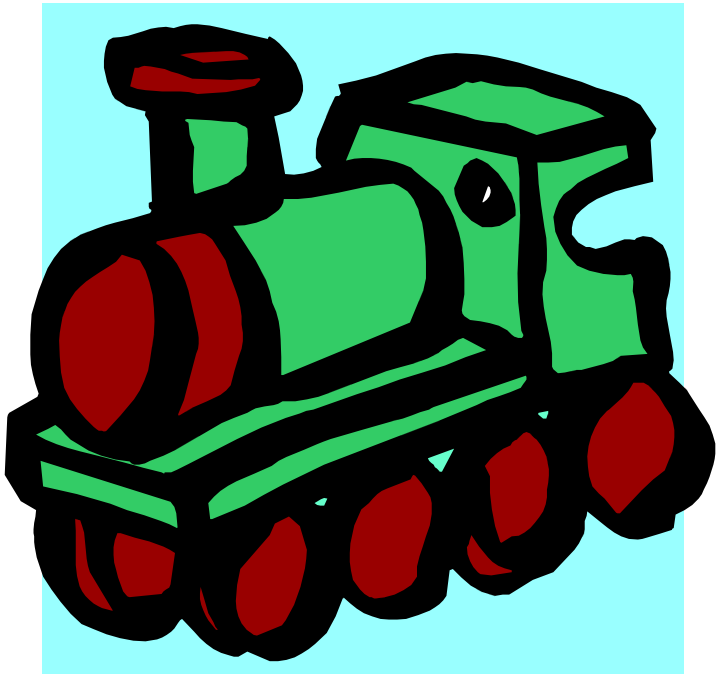
Entities



Entity Declaration

entity_declaration <=
    entity identifier **is**
        [**generic** (generic_interface_list);]
        [**port** (port_interface_list);]
        { entity_declarative_item }
    end [**entity**] [identifier] ;

interface_list <=
    identifier {, ... } : [mode]
      subtype_indication [:= expression] { ; ... }

Generics

- Erlauben die Beschreibung generischer (parametrisierter) Modelle
- Die *generic_interface_list* beschreibt die *generic constants*
- Beispiel:

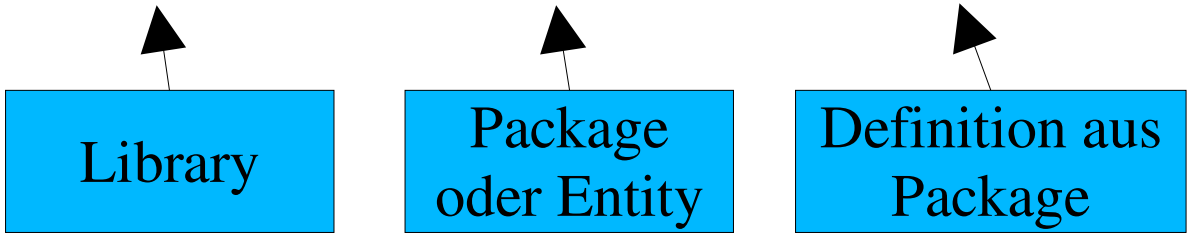
```
entity processor
   generic (cache_size : natural;
            bit_width  : natural := 32);
   port (...);
end processor;
```

Generics (2)

- Verwendung wie Konstanten:

```
architecture rtl of processor
signal cache : std_logic_vector
                    (cache_size-1 downto 0);
begin
    ...
g1: if cache_size /= 0 generate
        ...
    end generate g1;
    ...
end rtl;
```

Component Instantiation

- Soll eine Entity mit Generics instantiiert werden, so können vor den Ports die Generics angegeben werden
- ```
  p1 : entity processor
       generic map (cache_size => 1024,
                    bit_width  => 16)
       port map (...);
  ```
- Generics mit Default–Wert dürfen fehlen, Generics ohne Default–Wert müssen angegeben werden
  ```

Beispiel

```
entity reg is
   generic(width : positive);
   port(d : in bit_vector(0 to width-1);
        q : out bit_vector(0 to width-1);
        clk, reset : in bit);
end reg;
architecture rtl of reg is
begin
   p : process(clk, reset)
   begin
      if reset = '1' then
         q <= (others => '0');
      elsif clk'event and clk = '1' then
         q <= d;
end if; end process; end rtl;
```

Ports

- Ports können nicht nur
  - in,
  - out,
  - inout

  sein, sondern auch
  - buffer oder
  - linkage (falls das Modell in einer anderen Sprache gegeben ist).

Buffer Ports

- Ein Port vom Typ "buffer" ist wie ein "inout", es ist aber nur ein Treiber erlaubt
  - Deshalb kann immer der getriebene Wert gelesen werden
- Bei der Verwendung von Buffer Ports gibt es weitere Einschränkungen, unter anderem
  - Wird ein Buffer Port als "actual" an eine Komponente übergeben, so darf dieser Port nicht "out" sein
- Buffer Ports sollten vermieden werden

Analyse

- Jedes Modell ist eindeutig einer "Library" zugeordnet
- Ein Modell lässt sich eindeutig über das Triple Library – Entity – Architecture identifizieren
- Die aktuelle Library kann mittels "work" angesprochen werden
- Setzen der aktuellen Library ist in VHDL nicht möglich. Sie muss tool–spezifisch angegeben werden

Library Clauses

- Um eine bestimmte Library sichtbar zu machen: library <name>;
- Beispiel:

```
library ieee;
```

- Ansprechen einer Library:

```
library core;
...
inst: entity core.alu port map(...);
inst: entity alu port map(...);
```

Entity alu muss in library core definiert sein

Wählt Entity alu aus library "work" aus

Default Libraries

- Die beiden Libraries std und work stehen immer zur Verfügung
  - std enthält
    - Basistypen wie Bit, Boolean, Character und Operationen darauf, und
    - Funktionen zum Lesen und Schreiben aus Dateien (für den Simulator, nicht synthetisierbar!)
  - work bezieht sich auf die aktuelle Library

Use Clauses

- Direktes Sichtbarmachen von Komponenten einer Library
- Beispiel:

```
library ieee;
use ieee.std_logic_1164.std_ulogic;
```



- Die verwendete Library muss sichtbar sein
- Statt Package/Entity/Definition kann "**all**" stehen

Eindeutigkeit

- ```
  library core;
  use core.alu;
  ...
  inst: entity alu port map(...);
  ```

- ist illegal, wenn Entity alu sowohl in core als auch in "work" definiert ist (und work und core verschieden sind)
  ```

Packages

Einleitung

- Beim Entwurf großer Schaltungen ist eine Strukturierung notwendig, um
  - die Übersicht zu behalten
  - Teile mehrfach verwenden zu können
  - Teile getrennt synthetisieren / verifizieren zu können
- VHDL bietet dazu die Gliederung in
  - Libraries
  - Packages

Packages

- Packages dienen zur Gruppierung bestimmter Deklarationen, etwa bestimmter
  - Datentypen,
  - Unterprogramme,
  - Konstanten, usw.
- Trennung zwischen
  - der externen Sicht (*package declaration*) und
  - der Implementierung (*package body*)

Package Declaration

- **package** identifier **is**
  { package_declarative_item }
  end [identifier] ;
- Mögliche Deklarationen:
  - type, subtype
  - constant
  - signal
  - subprogram
  - components

Beispiel

```
package std_logic_1164 is
   type std_ulogic is ('U', 'X', '0', '1',
                       'Z', 'W', 'L', 'H', '-');
   ...
   function resolved (s : std_ulogic_vector)
                      return std_ulogic;
   subtype std_logic is resolved std_ulogic;
   subtype UX01 is resolved std_ulogic
         range 'U' TO '1'; -- ('U','X','0','1')
   ...
   function "not"(l : std_ulogic) return UX01;
   ...
end std_logic_1164;
```

Beispiel 2

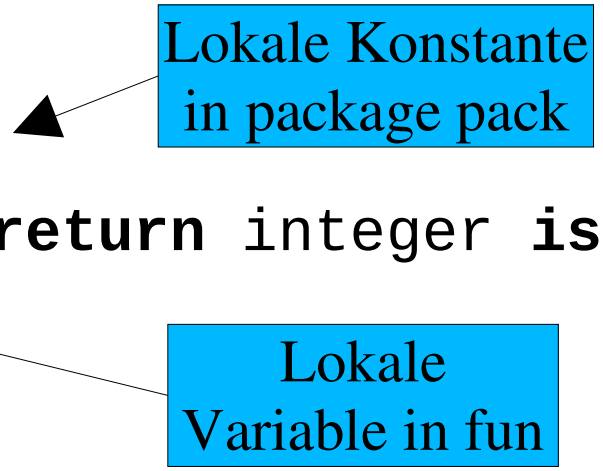
```
library ieee;
use ieee.std_logic_1164.all;

package cpu_types is
   constant word_size : positive := 16;
   constant address_size : positive := 24;
   subtype word is std_logic_vector
                   (word_size-1 downto 0);
   subtype address is std_logic_vector
                   (address_size-1 downto 0);
   type status_value is ( halted, idle, fetch,
              mem_read, mem_write, int_ack );
end cpu_types;
```

Deferred Constants

- Konstanten können ohne Wert deklariert werden:
```
constant max_buffer_size : positive;
```
- Der tatsächliche Wert ist versteckt und erst im Package Body angegeben
- Ist von Vorteil bei großen Projekten
- Achtung: es erfolgt keine optimierte Auswertung der Konstanten bei der Compilation

Package Body

```
package pack is
   function fun(a : integer) return integer;
end pack;
package body pack is
   constant c : integer := 7;
   function fun(a : integer) return integer is
      variable v : integer;
   begin
      v := a + c;
      return v;
   end;
end pack;
```



Packages und Synthese

- Packages sollten gemeinsam genutzte Typen, Konstanten, Unterprogramme enthalten
- Die Deklaration sollte keine internen Objekte enthalten
- Deferred Constants und globale Signale sollten nicht verwendet werden

Components

Components

- Komponenten eines Designs können durch Entities deklariert werden
- Alternativ können sie durch eine "*component declaration*" am Anfang einer Architecture angegeben werden
- Unterschiedliche Gesichtspunkte:
  - Entity beschreibt ein "echtes" Modul
  - Component beschreibt ein "virtuelles" oder "idealisiertes" Modul

Beispiel

- Es soll ein serielles Interface für einen Mikrokontroller entworfen werden
- Dieses soll von anderen Designern verwendet werden können
- Der Entwurf enthält
  - Ein Package, das die Sicht nach außen beschreibt
  - Die Entity des Interfaces
  - Die Implementierung (Architecture)

Beispiel: Package

```
library ieee;
use ieee.std_logic_1164.all;

package si_defs is
   subtype data_vector is
                  std_logic_vector(7 downto 0);
   constant data_size : positive := 8;
   ...
   component serial_interface
      port(clk : in std_logic;
           data : inout data_vector;
           ...);
end si_defs;
```

Beispiel: Entity

```
library ieee;
use ieee.std_logic_1164.all;
use work.si_defs.all;

entity serial_interface is
    port(clk : in std_logic;
         data : inout data_vector;
         ...);
end serial_interface;
```

Beispiel: Verwendung

```
library ieee;
use ieee.std_logic_1164.all;
use work.si_defs.serial_interface;

architecture structure of micro_controller is
begin
   ...
serial_a: serial_interface
          port map(clk => phi;
                   data => internal_data_bus;
                   ...);
   ...
end structure;
```

Configurations

Binding

- Welche Implementierung (Entity, Architecture) soll bei einer Komponenteninstantiierung verwendet werden?
  - Configuration
  - Default Binding

Configuration Declaration (einfach)

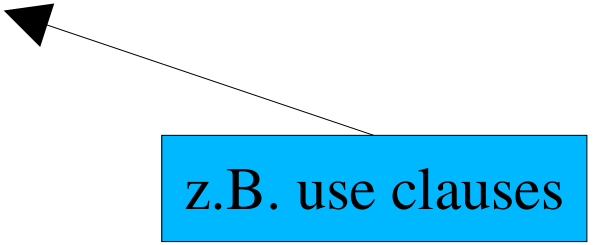
- Eigenständige Deklaration, wie Entity oder Package
- **configuration** identifier **of** *entity_name* **is**
  for *architecture_name*
  for *instantiation_label* : *component_name*
  use entity *entity_name* ;
  end for ;
  end for;
  end [identifier] ;

Beispiel: serielles Interface

```
architecture structure of micro_controller is
...
serial_a: serial_interface port map(...);
...
end structure;

configuration m_conf of micro_controller is
   for structure
      for serial_a : serial_interface
         use entity
                work.si_defs.serial_interface;
      end for; -- of serial_a
   end for; -- of architecture structure
end m_conf;
```

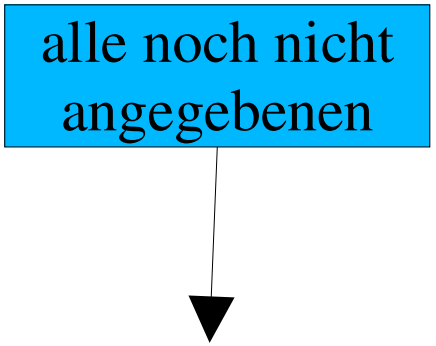
Configuration (komplett)

- Konfiguration kann über mehrere Hierarchiestrukturen erfolgen
- **configuration** identifier **of** *entity_name* **is**
  { configuration_declarative_item }
  block_configuration
  end [identifier] ;



Configuration (komplett) (2)

- block_configuration <=
  for block_specification
  { configuration_item }
  end for ;
- block_specification <=
  architecture_name | statement_label | **others** | **all**
- configuration_item <=
  block_configuration | binding_indication



Binding Indication

- binding_indication <=
  use entity *entity_name*
  | **use entity** *entity_name* (*architecture_name*)
  | **use configuration** *configuration_name* ;

Beispiel

```
library xy_lib;
use xy_lib.dff;
configuration full of counter is
   for rtl -- architecture of counter
      for all : digit_register
         use entity work.reg4(struct);
         for struct -- architecture of reg4
            for bit0 : flipflop
               use entity dff(hi_fanout);
            end for;
            for others : flipflop
               use entity dff(basic);
            end for;
         end for; -- architecture struct
      end for;
      ...  -- binding for other comp. instances
   end for; -- architecture rtl
end full;
```

Direct Instantiation of Configured Entities

- Die Instantiierung einer Component kann auch die folgende Gestalt haben:
  component_instantiation_statement <=
  label :
  configuration configuration_name
  [**generic map** (generic_association_list)]
  [**port map** (port_association_list)] ;
- Die Configuration wird direkt angegeben
- Erst ab VHDL93 erlaubt

Konfigurationen und Synthese

- Konfigurationen sollten verwendet werden, falls mehr als eine Architecture zu einer Entity existiert
- Namen von Entity und Component sollten übereinstimmen
- Konfigurationen sollten einfach gehalten werden und nie über mehrere Hierarchiestufen gehen. Stattdessen können mehrere Konfigurationen verwendet werden

Default Binding

- Wenn keine "Configuration" angegeben ist, wird ein *default binding* verwendet
  - Komponenten werden mit Entities gleichen Namens verbunden
    - **Achtung**: Die Entities müssen dazu sichtbar sein, etwa mit "use library.dff;"
  - Für Entities wird die zuletzt übersetzte Architecture ausgewählt
    - Obwohl diese dadurch in der Regel genau bestimmt ist, sollte man sich darauf nicht verlassen (fehlende Übersichtlichkeit!)

Deklarationsmöglichkeiten

Deklaration	Entity	Archit./ Block	Process/ Subprg.	Package Declarat.	Package Body
Signal	✓	✓	–	✓	–
Variable	–	–	✓	–	–
Konstante	✓	✓	✓	✓	✓
Component	–	✓	–	✓	–
Subprogram	✓	✓	✓	✓	✓
Subprg. Body	✓	✓	✓		✓
Type/Subtype	✓	✓	✓	✓	✓
Configuration	–	✓	–	–	–
Use Clause	✓	✓	✓	✓	✓

Die Bibliothek

IEEE

Die Bibliothek "IEEE"

- Dieses Paket ist von IEEE standardisiert worden (Standard 1164)
- Industrie–weiter Standard

Das Package std_logic_1164

- Datentyp std_(u)logic und std_(u)logic_vector
- Subtypen:

```
subtype X01 is
      resolved std_ulogic range 'X' to '1';
subtype X01Z is
      resolved std_ulogic range 'X' to 'Z';
subtype UX01 is
      resolved std_ulogic range 'U' to '1';
subtype UX01Z is
      resolved std_ulogic range 'U' to 'Z';
```

- Operatoren and, nand, or, nor, xor, xnor, not auf std_(u)logic und std_(u)logic_vector

std_logic_1164 (cont.)

- Conversion functions

```
function to_bit(s : std_ulogic;
                  xmap : bit := '0')
                return bit is
begin
   case s is
   when '0' | 'L' => return ('0');
   when '1' | 'H' => return ('1');
   when others    => return xmap;
   end case;
end;
```

std_logic_1164 (cont.)

- Conversion functions (cont.)

```
function to_bit(s : std_ulogic;
                xmap : bit := '0')
         return bit;
function to_bitvector(s : std_ulogic_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 to_stdULogicvector(b : bit_vector)
         return std_ulogic_vector;
```

std_logic_1164 (cont.)

- Conversion functions (cont.)

```
function To_X01  ( s : std_logic_vector  )
  return std_logic_vector;
function To_X01  ( s : std_ulogic_vector )
  return std_ulogic_vector;
function To_X01  ( s : std_ulogic        )
  return X01;
function To_X01  ( b : bit_vector        )
  return std_logic_vector;
function To_X01  ( b : bit_vector        )
  return std_ulogic_vector;
function To_X01  ( b : bit               )
  return X01;
```

- Analog für To_X01Z und To_UX01

std_logic_1164 (cont.)

- Edge detection

```
function rising_edge(signal s : std_ulogic)
  return boolean;
function falling_edge(signal s : std_ulogic)
  return boolean;
```

- Object contains an unknown

```
function is_X(s : std_ulogic_vector)
  return boolean;
function is_X(s : std_logic_vector)
  return boolean;
function is_X(s : std_ulogic)
  return boolean;
```

Das Package "std_logic_arith"

- ```
  type unsigned is array (natural range <>)
                   of std_logic;
  ```
- ```
  type signed   is array (natural range <>)
                   of std_logic;
  ```
- Operationen
  - "+", "–", "*", abs,
  - "<", "<=", ">", ">=", "=", "/="
  - shl, shr

std_logic_arith (cont.)

- Conversion functions
  - conv_integer (signed, unsigned, std_ulogic –> integer)
  - conv_signed (..., size –> signed)
  - conv_unsigned (..., size –> unsigned)
  - conv_std_logic_vector (..., size –> std_logic_vector)
  - ext (zero extend std_logic_vector)
  - sxt (sign extend std_logic_vector)

Synthese

- Die Packages ieee.std_logic_1164 und ieee.std_logic_arith sollten verwendet werden
  - Oft sind speziell optimierte Synthesefunktionen vorhanden
  - Bessere Lesbarkeit / Wartbarkeit