EE 459/500 – HDL Based Digital Design with Programmable Logic

Lecture 8
Packages and Libraries

Read before class:

Chapter 2 from textbook

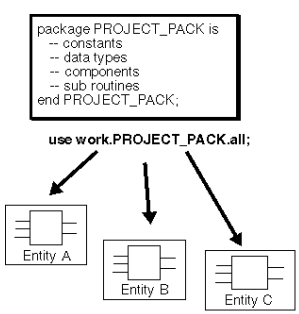
Overview

- Packages
- Libraries
- Testbenches
- Writing VHDL code for synthesis

VHDL Structural Elements

- **Entity:** description of interface consisting of the port list.
  - The primary hardware abstraction in VHDL, analogous to a symbol in a block diagram.
- **Architecture:** description of the function of the corresponding module.
- **Process:** allows for a sequential execution of the assignments
- **Configuration:** used for simulation purposes.
- **Package:** hold the definition of commonly used data types, constants and subprograms.
- **Library:** the logical name of a collection of compiled VHDL units (object code).
  - Mapped by the simulation or synthesis tools.

Packages Declaration



- Collection of definitions of constants, data types, components, and subprograms.
- A package serves as a central repository for frequently used utilities, such as component declarations.
- The declarations may then be reused by any VHDL model by simply accessing the package.
- **use WORK.PROJECT_PACK.all;**
  - The architecture accesses the component declarations in the package "PROJECT_PACK" located in the library WORK via the use clause.
  - Use clause placed just before the architecture body statement.

Example

```
package EXAMPLE_PACK is
     type SUMMER is (JUN, JUL, AUG);
     component D_FLIP_FLOP
          port ( D, CK : in BIT;
                 Q, QBAR : out BIT);
     end component;
     constant PIN2PIN_DELAY : TIME := 125ns;
     function INT2BIT_VEC (INT_VALUE : INTEGER)
          return BIT_VECTOR;
end EXAMPLE_PACK

library DESIGN_LIB;                -- library clause
use DESIGN_LIB.EXAMPLE_PACK.all; -- use clause

entity EXAM is ...
```

Package Body

- **Package body** is used to store the definitions of functions and procedures that were declared in the corresponding package declaration.
- A package body is always associated with a package declaration.
- **Example:**

```
package body EXAMPLE_PACK is
     function INT2BIT_VEC (INT_VALUE: INTEGER)
          return BIT_VECTOR is
     begin
          -- behavior of function described here
     end INT2BIT_VEC;
end EXAMPLE_PACK;
```

Example of Package

```
package LOGIC_OPS is -- package

-- Declare logic operators
component AND2_OP
   port (A, B : in BIT;
         Z : out BIT);
end component;

component OR3_OP
   port (A, B, C : in BIT;
         Z : out BIT);
end component;

component NOT_OP
    port (A : in BIT;
          A_BAR : out BIT);
end component;

end LOGIC_OPS;
```

Example of Package Usage

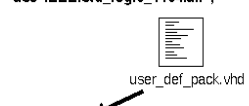
```
-- entity declaration
entity MAJORITY is
    port ( A_IN, B_IN, C_IN : in BIT;
           Z_OUT : out BIT);
end MAJORITY;

-- architecture description
-- uses components from package LOGIC_OPS in library WORK
use WORK.LOGIC_OPS.all;

architecture STRUCTURE of MAJORITY is
    signal  INT1, INT2, INT3 : BIT;
begin
    A1: AND2_OP port map (A_IN, B_IN, INT1);
    A2: AND2_OP port map (A_IN, C_IN, INT2);
    A3: AND2_OP port map (B_IN, C_IN, INT3);
    O1: OR3_OP port map (INT1, INT2, INT3, Z_OUT);
end STRUCTURE;
```

Overview

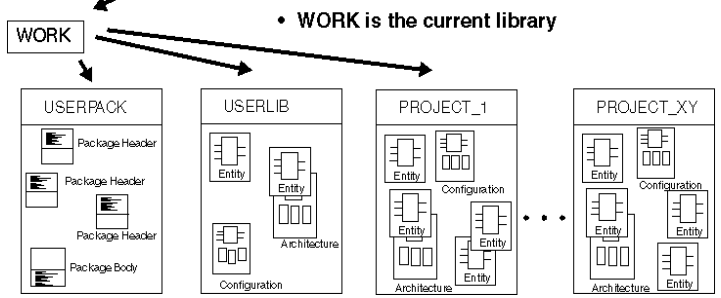
- Packages
- Libraries
- Testbenches
- Writing VHDL code for synthesis

Library

```
library IEEE ;
use IEEE.std_logic_1164.all ;
```



- Collection of compiled design units
- Physical location of the compiled VHDL data file entity, architecture, package header and body, configuration
- WORK is the current library



Library

- There are two reserved library names always available to designers.
  - **WORK**: Predefined library
  - **STD:** The **STD** contains two packages: **STANDARD** provides declaration for predefined types (real, integers, Boolean, etc). **TEXTIO** contains useful subprograms that enables to perform ASCII file manipulations
    - **library** STD; -- declares STD to be a library
    - **use** STD.STANDARD.**all;** -- use all declarations in package
      -- STANDARD, such as BIT, located
      -- within the library STD
- A library clause declares **WORK** to be a library.
  library WORK; -- WORK is predefined library

Library

- User-defined libraries: LOGIC_LIB
  - Assume the LOGIC_OPS package is located in the LOGIC_LIB library instead of WORK library.

```
library LOGIC_LIB;
use LOGIC_LIB.LOGIC_OPS.all;
architecture STRUCTURE of MAJORITY is
-- use components in package LOGIC_OPS of library
   LOGIC_LIB
```

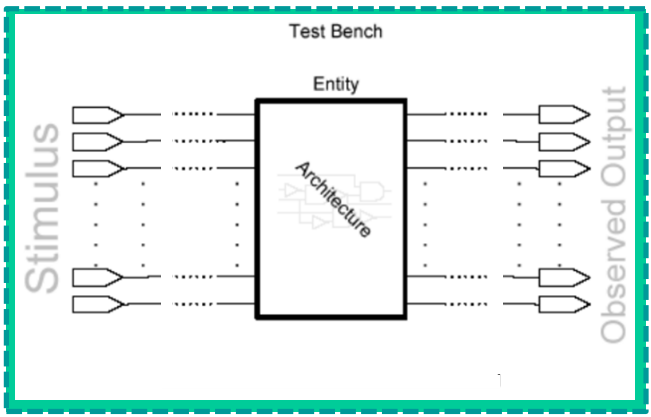
Overview

- Packages
- Libraries
- Testbenches
- Writing VHDL code for synthesis

Testbench

- Used to verify the specified functionality of a design
  - Provides the stimuli (test vectors) for the Unit Under Test (UUT), analyzes the UUT's response or stores the values in a file.
  - Simulation tools visualize signals by means of a waveform which the designer compares with the expected response. Debug if does not match.
- Does not need to be synthesizable
- No ports to the outside, self-contained

Testbench



Testbench

- Simple testbench responses can be analyzed by waveform inspection
- Sophisticated testbenches may require more complicated verification techniques
  - Can take >50% of project resources
  - Do not underestimate the value/importance of testbenches!

Structure of a VHDL Testbench

```
entity TB_TEST is
end TB_TEST;

architecture BEH of TB_TEST is
  -- component declaration of UUT
  -- internal signal definition
begin
  -- component instantiation of UUT

  -- clock and stimuli generation
  wait for 100 ns;
  A <= 0;
  CLK <= 1;
  …
end BEH;

configuration CFG1 of TB_TEST is
   for BEH;
       -- customized configuration
   end for;
end CFG_TB_TEST;
```

- Declaration of the **Unit Under Test (UUT)**
- Connection of the UUT with testbench signals
- Stimuli and clock generation (behavioral modeling)
- Response analysis
- A **configuration** is used to pick the desired components for simulation
  - May be a customized configuration for testbench simulation

Example: Simple Testbench

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

entity ADDER is
   port (A,B : in bit;
      CARRY,SUM : out bit);
end ADDER;


architecture RTL of ADDER is
begin
  ADD: process (A,B)
  begin
     SUM <= A xor B;
     CARRY <= A and B;
  end process ADD;
end RTL;
```

```
entity TB_ADDER IS -- empty entity is defined
end TB_ADDER;      -- no need for interface

architecture TEST of TB_ADDER is
   component ADDER
     port (A, B: in bit;
           CARRY, SUM: out bit);
   end component;
   signal A_I, B_I, CARRY_I, SUM_I : bit;

begin
  UUT: ADDER port map(A_I, B_I, CARRY_I, SUM_I);

  STIMULUS: process
    begin
       A_I <= 0 ; B_I <= 0 ; wait for 10 ns;
       A_I <= 1 ; B_I <= 1 ; wait for 10 ns;
       A_I <= 1 ; B_I <= 0 ; wait for 10 ns;
       A_I <= 1 ; B_I <= 1 ; wait for 10 ns;
       wait;
      -- and so on ...
  end process STIMULUS;
end TEST;

configuration CFG_TB_ADDER of TB_ADDER is
      for TEST
      end for;
end CFG_TB_ADDER;
```

Configuration

- A VHDL description may consist of many design entities, each with several architectures, and organized into a design hierarchy. The **configuration** does the job of specifying the exact set of entities and architectures used in a particular simulation or synthesis run.

- A configuration does two things:
  1) A configuration specifies the design entity used in place of each component instance (i.e., it plugs the chip into the chip socket and then the socket-chip assembly into the PCB).
  2) A configuration specifies the architecture to be used for each design entity (i.e., which die).

Configuration

- A configuration statement is used to bind a component instance to an entity-architecture pair. A configuration can be considered as a parts list for a design. It describes which behavior to use for each entity, much like a parts list describes which part to use for each part in the design.
- Component configuration can be performed outside the architecture body which instantiates a certain component.
- A configuration declaration is a design unit which can be compiled separately.
- The particular architecture body has not to be recompiled when the binding is changed.
- See detailed discussion in Appendix B.

Example

```
use WORK.all;
architecture PARITY_STRUCTURAL of PARITY is
  component XOR_GATE --component declaration
    port(X,Y: in BIT; Z: out BIT);
  end component;
  component INV --component declaration
    port(X: in BIT; Z: out BIT);
  end component;
  signal T1, T2, T3: BIT;
begin
  XOR1: XOR_GATE port map (V(0), V(1), T1);
  XOR2: XOR_GATE port map (V(2), V(3), T2);
  XOR3: XOR_GATE port map (T1, T2, T3);
  INV1: INV port map (T3, EVEN);
end PARITY_STRUCTURAL;
```

```
use WORK.all;
configuration CONFIG_1 of PARITY is
  for PARITY_STRUCTURAL
    for XOR1,XOR2:XOR_GATE use
      entity XOR_GATE(ARCH_XOR_1);
    end for;
    for XOR3:XOR_GATE use
      entity XOR_GATE(ARCH_XOR_2);
    end for;
    for INV1:INV use
      entity INV(ARCH_INV_1);
    end for;
  end for;
end CONFIG_1;
```

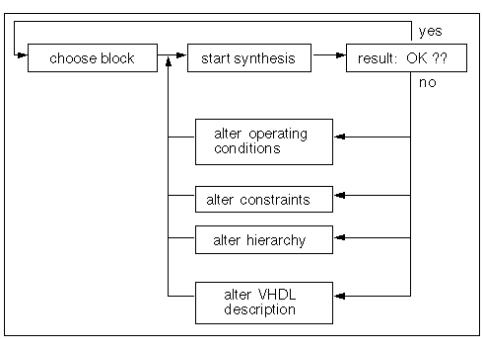
Overview

- Packages
- Libraries
- Testbenches
- Writing VHDL code for synthesis

How to write good VHDL code with Synthesis in mind?

- Constraints
  - Speed
  - Area
  - Power
- Macrocells
  - Adder
  - Comparator
  - Bus interface
- Optimizations
  - Boolean: mathematical
  - Gate: technological
  - The **optimization phase** requires quite a lot of iterations before the software reports its final result.

Synthesis Process in Practice



- In most cases synthesis has to be carried out several times in order to achieve an optimal synthesis result

Write code for Synthesis: Guidelines

- Consider the effects of **different coding styles** on the inferred hardware structures
  - If Then Else vs. Case vs. ...
- Appropriate design partitioning
  - Critical paths should not be distributed to several synthesis blocks
  - Automatic synthesis performs best at module sizes of several 1000 gates
  - Different optimization constraints used for separate blocks
    - High speed parts can be synthesized with very stringent timing constraints
    - Non-critical parts should consume the least amount of resources (area) possible.

Combinational Process

```
-- Example: multiplexer
process (A, B, SEL)
begin
  if (SEL = '1') then
      OUT <= A;
  else
      OUT <= B;
  end if;
end process
```

- In simulation, a process is activated when an event occurs on one of its signals from the sensitivity list.
- Sensitivity list is usually ignored during synthesis .
- Equivalent behavior of simulation model and hardware: sensitivity list must contain all signals that are read by the process.

If the signal SEL was missing in the process sensitivity list, synthesis would create exactly the same result, namely a multiplexer, but simulation will show a completely different behavior!

Incomplete Assignment

```
Library IEEE;
use IEEE.Std_Logic_1164.all;

entity INCOMP_IF is
   port (A, SEL: in std_logic;
         Z: out std_logic);
end INCOMP_IF;

architecture RTL of INCOMP_IF is
begin
 process (A, SEL)
   begin
     if SEL = '1' then
         Z <= A;
     end if;
   end process;
end RTL;
```

- What is the value of Z, if SEL = `0`?
  - The old value of Z will be maintained in the simulation, that means no change will be carried out on Z.
- What hardware would be generated during synthesis?
  - The synthesis tools creates a **latch**, in which the SEL signal is connected as the clock input. It is an element very difficult to test in the synchronous design, and therefore it **should not be used**.

Rules for Synthesizing Combinational Logic

- Complete sensitivity list
  - RTL behavior has to be identical with hardware realization
  - An incomplete sensitivity list can cause warnings or errors
- No incomplete IF-statements are allowed
  - Because they result in transparent latches

Combinational Logic

```
architecture EXAMPLE of FEEDBACK is
  signal B,X : integer range 0 to 99;
begin
   process (X, B)
   begin
      X <= X + B;
   end process;

   . . .
end EXAMPLE;
```

- **Do not create combinational feedback loops!**
  - A feedback loop triggers itself all the time.
  - X is increased to its maximum value. So simulation quits at time 0 ns with an error message because X exceeds its range.

Coding Style Influence

Direct Implementation

```
process (SEL,A,B)
begin

  if SEL = `1` then
       Z <= A + B;
  else
       Z <= A + C;
  end if;
end process;
```

Manual resource sharing

```
process (SEL,A,B)
variable TMP : bit;
begin
   if SEL = `1` then
        TMP := B;
   else
        TMP := C;
   end if;
   Z <= A + TMP;
end process;
```

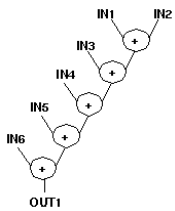
Manual resource sharing is recommended as it leads to a better starting point for the synthesis process.

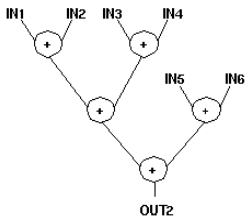
Adder is shared!

Source Code Optimization

An operation can be described very efficiently for synthesis:

```
OUT1 <= IN1+IN2+IN3+IN4+IN5+IN6    OUT2 <= (IN1+IN2)+(IN3+IN4)+(IN5+IN6)
```





- In one description the longest path goes via five, in the other description via three addition components - some optimization tools automatically change the description according to the given constraints.

Summary

- Packages and libraries are useful for code re-use
- Testbenches are crucial in the initial phase of VHDL code writing and design debug via simulation
- Arriving to a good VHDL coding style (for synthesis!) requires practice, practice, practice = experience. Start with understanding and honoring the provided guidelines.

Appendix A: Assignment with Array Types

- Elements are assigned according to their position, not their number
- The direction of arrays should always be defined the same way

```
architecture EXAMPLE of ARRAYS is
 signal Z_BUS : bit_vector (3 downto 0);
 signal C_BUS : bit_vector (0 to 3);
begin
   Z_BUS <= C_BUS;
end EXAMPLE;
```

Slices of Arrays

- Slices select elements of arrays

```
architecture EXAMPLE of SLICES is
  signal BYTE : bit_vector (7 downto 0);
  signal A_BUS, Z_BUS : bit_vector (3 downto 0);
  signal A_BIT : bit;

begin
  BYTE (5 downto 2) <= A_BUS;
  BYTE (5 downto 0) <= A_BUS; -- wrong

  Z_BUS (1 downto 0) <= `0` & A_BIT;
  Z_BUS <= BYTE (6 downto 3);
  Z_BUS (0 to 1) <= `0` & B_BIT; -- wrong
  A_BIT <= A_BUS (0);
end EXAMPLE;
```

The direction of the "slice" and of the "array" must match!

Aggregates

```
architecture EXAMPLE of AGGREGATES is

   signal BYTE : bit_vector (7 downto 0);
   signal Z_BUS : bit_vector (3 downto 0);
   signal A_BIT, B_BIT, C_BIT, D_BIT : bit;

begin
  Z_BUS <= ( A_BIT, B_BIT, C_BIT, D_BIT ) ;
  ( A_BIT, B_BIT, C_BIT, D_BIT ) <= bit_vector'("1011");
  ( A_BIT, B_BIT, C_BIT, D_BIT ) <= BYTE(3 downto 0);
  BYTE <= (7 => '1', 5 downto 1 => '1', 6 => B_BIT, others => '0');
end EXAMPLE;
```

- Aggregates bundle signals together, may be used on both sides of an assignment
- keyword **'others'** selects all remaining elements
- Some aggregate constructs may not be supported by your synthesis tool

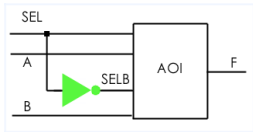
Appendix B: More on Configuration

- Analogy with a PCB:
  - Design entity (chip package) consists of both an entity declaration (chip pins) and architecture body (chip die)
- Configurations:
  - Select one architecture from many architectures of one design entity for instantiation (i.e., specify which die goes in the package of the chip that will be plugged into the PCB)
  - Choose from amongst different design entities for instantiation (essentially specifying which chip to plug into the socket)

Components and port maps

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity MUX2 is
  port (SEL, A, B: in STD_LOGIC;
  F : out STD_LOGIC);
end;
```

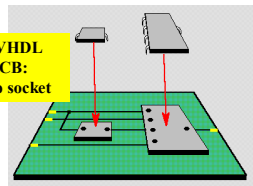
entity declaration = chip pins



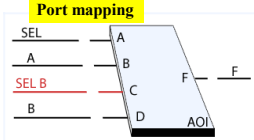
architecture declaration = chip die

```
architecture STRUCTURE of MUX2 is
component INV
  port (A: in STD_LOGIC;
  F: out STD_LOGIC);
end component;
component AOI
  port (A, B, C, D: in STD_LOGIC;
  F : out STD_LOGIC);
end component;
signal SELB: STD_LOGIC;

begin
  G1: INV port map (SEL, SELB);
  G2: AOI port map (SEL, A, SELB, B, F);
end;
```

Instantiating components in VHDL is like plugging chips into a PCB: component declaration = chip socket



Port mapping



Configuration

Default configuration of MUX2

```
use WORK.all;
configuration MUX2_default_CFG of MUX2 is
  for STRUCTURE
    -- Components inside STRUCTURE configured by default
    -- let's say v2 architecture for AOI
  end for;
end MUX2_default_CFG;
```

Specified configuration of MUX2

```
use WORK.all;
configuration MUX2_specified_CFG of MUX2 is
  for STRUCTURE
    for G2 : AOI
      use entity work.AOI(v1);
      -- architecture v1 specified for AOI design entity
    end for;
  end for;
end MUX2_specified_CFG;
```

Configuration

- Default binding configuration:
  - The chip socket (component declaration) carries a chip (design entity) of the same name (e.g.,, AOI). The chip is inserted into the socket courtesy of a component instantiation and a configuration declaration. If configuration is omitted or if we use a default configuration, the socket and chip must have the same name.
- Specified configuration:
  - If we want to choose a particular die (architecture) for our chip, we must specify the architecture in the configuration.
- Late-binding configuration:
  - Suppose we want to create a general-purpose socket and at some later time, we want to specify which chip will be plugged into the socket. To do this requires a late-binding configuration declaration.

Configuration

Late-binding configuration of MUX2

```
use WORK.all;
configuration AND4_CFG of MUX2 is
  for STRUCTURE
    for G2 : AOI
      use entity work.AND4(quick_fix);
      -- architecture quick_fix of AND4 specified for AOI component
    end for;
  end for;
end AND4_CFG;
```