

In this lecture we will deepen our knowledge and skills regarding practical hardware design using VHDL and structural modelling.

HWMoD
WS24

Struct. Mod.
Introduction
Instances
Components
Architecture
Selection

Hardware Modeling [VU] (191.011) – WS24 – Structural Modeling

Florian Huemer & Sebastian Wiedemann & Dylan Baumann

WS 2024/25

Structural Modeling

Introduction

Introduction

	Behavior	Structure	Geometry
System Level	Block Diagram Data Flow		
Algorithmic Level	Algorithm Control Flow		
Register Transfer Level (RTL)	Register File Data Path Control		
Logic Level	Logic Function Data Path		
Circuit Level	Transistor Model Data Path		

A fundamental aspect of any hardware description language is its support for hierarchical design, enabling the creation of complex circuits by assembling simpler building blocks into larger systems. To understand the significance of this, imagine an object-oriented programming language that prohibits objects from containing other objects. Obviously, such a language would be severely limited in its utility. This design approach corresponds to the logic and register-transfer levels on the structural axis of the Y-Diagram, which is why it is called "structural modeling".

Introduction

HWMoD
WS24

Struct. Mod.
Introduction
Instances
Components
Architecture
Selection

	Behavior	Structure	Geometry
System Level	Inputs : Keyboard Output: Display Funktion:		
Algorithmic Level	while input Read „Schilling“ Calculate Euro Display „Euro“		
Register Transfer Level (RTL)	if A = '1' then B := B + 1 else B := B end if		
Logic Level	D = NOT E C = (D OR B) AND A		
Circuit Level	$\frac{dU}{dt} = R \frac{dI}{dt} + \frac{1}{C} + L \frac{d^2I}{dt^2}$		

Hence, structural modeling can be defined as the process of combining and interconnecting modules to create more complex modules in a hierarchical fashion. Including an entity as a sub-module in another design unit in VHDL is referred to as instantiation – the included sub-module being the instance. At this point, let us also define some other commonly used terms in hardware design and testing related to instantiations that will be used throughout this course.

Introduction (cont'd)

HWMoD
WS24

Struct. Mod.
Introduction
Instances
Components
Architecture
Selection

Structual Modeling

Create complex modules by combining and interconnecting (simpler) sub-modules.

The top-level entity or top-level design refers to the highest or outermost module in a hierarchical design. It is the main component that integrates all the lower-level modules and subcomponents. This top-level entity serves as the entry point for the design and defines the inputs and outputs that connect the circuit to external systems.

Introduction (cont'd)

HWMoD
WS24

Struct. Mod.
Introduction
Instances
Components
Architecture
Selection

Structual Modeling

Create complex modules by combining and interconnecting (simpler) sub-modules.

Top-Level Design/Entity

The design unit (entity) that sits on highest layer of a hierarchical hardware design.

Another term that you will also hear quite often is Unit-Under-Test or UUT. It refers to the entity that is instantiated in a testbench to validate its functionality.

Introduction (cont'd)

HWMoD
WS24

Struct. Mod.
Introduction
Instances
Components
Architecture
Selection

Structual Modeling

Create complex modules by combining and interconnecting (simpler) sub-modules.

Top-Level Design/Entity

The design unit (entity) that sits on highest layer of a hierarchical hardware design.

Unit Under Test (UUT)

The module instantiated in a testbench and whose behavior is verified.

Structural Modeling

Instances

Creating Instances – Example: Half Adder



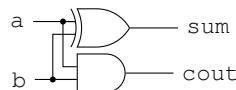
```

1 entity xor_gate is
2   port (
3     a, b : in std_ulogic;
4     x : out std_ulogic
5   );
6 end xor_gate;
7
8 architecture arch of xor_gate is
9   signal x_xor : std_ulogic;
10  x_xor <= a xor b;
11 end architecture;
12
13 entity and_gate is
14   port (
15     a, b : in std_ulogic;
16     x : out std_ulogic
17   );
18 end and_gate;
19
20 architecture arch of and_gate is
21   signal x_and : std_ulogic;
22  x_and <= a and b;
23 end architecture;

```

Before we go into details on the formal syntax specification of instantiations in VHDL, let us first look at a simple code example. Consider the half adder circuit shown on this slide, which is used to calculate the sum and carry of two single-bit inputs and consists of a single "XOR" and "AND"-gate. We want to construct this circuit out of the two entities shown on the slide. These two modules should already look quite familiar to you, as they simply use one concurrent signal assignment to implement their desired behaviors. Also notice that we used the `std_ulogic` datatype for its inputs and outputs.

Creating Instances – Example: Half Adder



```

1 entity xor_gate is
2   port (
3     a, b : in std_ulogic;
4     x : out std_ulogic
5   );
6 end entity;
7
8 architecture arch of xor_gate is
9 begin
10  x <= a xor b;
11 end architecture;

```

```

1 entity and_gate is
2   port (
3     a, b : in std_ulogic;
4     x : out std_ulogic
5   );
6 end entity;
7
8 architecture arch of and_gate is
9 begin
10  x <= a and b;
11 end architecture;

```

Structural Modeling

Instances

Creating Instances – Example: Half Adder

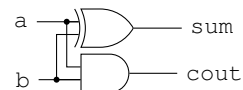


Here, we see an implementation of the half adder circuit using instances of the entities shown on the previous slides.

Creating Instances – Example: Half Adder

```

1 entity ha is
2   port (
3     a, b : in std_ulogic;
4     sum, cout : out std_ulogic
5   );
6 end entity;
7
8 architecture arch of ha is
9   begin
10    and_gate_inst : entity work.and_gate
11    port map (a, b, cout);
12
13    xor_gate_inst : entity work.xor_gate
14    port map (a, b, sum);
15 end architecture;
```



- Instances → statement part
- Two instantiation statements
- Positional association
- Inputs must be readable, outputs writable

Structural Modeling

Instances

Creating Instances – Example: Half Adder



- Instances → statement part
- Two instantiation statements
- Positional association
- Inputs must be readable, outputs writable

Its entity declaration lists the inputs `a` and `b` as well as the outputs `sum` and `cout`.

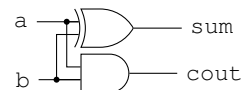
Creating Instances – Example: Half Adder

HWMMod
WS24

Struct. Mod.
Introduction
Instances
Port Map
Unused Ports
Generic Map
Components
Architecture
Selection

```

1 entity ha is
2   port (
3     a, b : in std_ulogic;
4     sum, cout : out std_ulogic
5   );
6 end entity;
7
8 architecture arch of ha is
9   begin
10    and_gate_inst : entity work.and_gate
11    port map (a, b, cout);
12
13    xor_gate_inst : entity work.xor_gate
14    port map (a, b, sum);
15 end architecture;
```

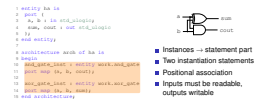


- Instances → statement part
- Two instantiation statements
- Positional association
- Inputs must be readable, outputs writable

Structural Modeling

Instances

Creating Instances – Example: Half Adder



As already discussed in a previous lecture, in VHDL instances are created in the statement part of architectures.

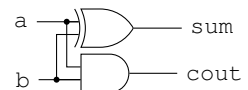
Creating Instances – Example: Half Adder

HWMoD
WS24

Struct. Mod.
Introduction
Instances
Port Map
Unused Ports
Generic Map
Components
Architecture
Selection

```

1 entity ha is
2   port (
3     a, b : in std_ulogic;
4     sum, cout : out std_ulogic
5   );
6 end entity;
7
8 architecture arch of ha is
9   begin
10    and_gate_inst : entity work.and_gate
11    port map (a, b, cout);
12
13    xor_gate_inst : entity work.xor_gate
14    port map (a, b, sum);
15 end architecture;
```



- Instances → statement part
- Two instantiation statements
- Positional association
- Inputs must be readable, outputs writable

Structural Modeling

Instances

Creating Instances – Example: Half Adder



To do so, one has to select a name for the instance followed by a colon and the name of the object that should be instantiated.

Creating Instances – Example: Half Adder

HWMMod
WS24

Struct. Mod.

Introduction

Port Map

Unused Ports

Generic Map

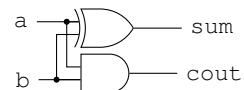
Components

Architecture

Selection

```

1 entity ha is
2   port (
3     a, b : in std_ulogic;
4     sum, cout : out std_ulogic
5   );
6 end entity;
7
8 architecture arch of ha is
9 begin
10  and_gate_inst : entity work.and_gate
11    port map (a, b, cout);
12
13  xor_gate_inst : entity work.xor_gate
14    port map (a, b, sum);
15 end architecture;
```



- Instances → statement part
- Two instantiation statements
- Positional association
- Inputs must be readable, outputs writable

Structural Modeling

Instances

Creating Instances – Example: Half Adder



Then the port map statement is used to connect the interface signals of the instance to local signals available in the architecture.

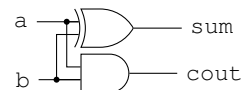
Creating Instances – Example: Half Adder

HWMMod
WS24

Struct. Mod.
Introduction
Instances
Port Map
Unused Ports
Generic Map
Components
Architecture
Selection

```

1 entity ha is
2   port (
3     a, b : in std_ulogic;
4     sum, cout : out std_ulogic
5   );
6 end entity;
7
8 architecture arch of ha is
9   begin
10    and_gate_inst : entity work.and_gate
11      port map (a, b, cout);
12
13    xor_gate_inst : entity work.xor_gate
14      port map (a, b, sum);
15 end architecture;
```



- Instances → statement part
- Two instantiation statements
- Positional association
- Inputs must be readable, outputs writable

Structural Modeling

Instances

Creating Instances – Example: Half Adder



For our half adder example circuit we create two instances – one of each of the basic gates shown on the previous slide. The local signals in the half adder architecture are connected to the interface signals of the entities using positional association. This works exactly like with aggregate expressions – already introduced in a previous lecture to initialize records. The signals are simply connected in the sequence of how they appear in the entity declaration of the instantiated entities. For our example this means that the local signal `a` is connected to the ports named `a` of the instances, just as `b` is connected to `b` inputs. The output `x` of the and gate is connected to `cout`, while the XOR's output is connected to the `sum` signal. Of course it must be ensured that signals that are connected to outputs of an instance are actually writable in the instantiation architecture. Similarly, input signals must be readable. It would, for example, lead to a compilation error if the `x` output of one of the gates would be connected to the `a` input of the half adder.

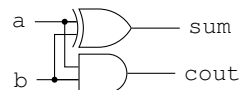
Creating Instances – Example: Half Adder

HWMoD
WS24

Struct. Mod.
Introduction
Instances
Port Map
Unused Ports
Generic Map
Components
Architecture
Selection

```

1 entity ha is
2   port (
3     a, b : in std_ulogic;
4     sum, cout : out std_ulogic
5   );
6 end entity;
7
8 architecture arch of ha is
9   begin
10    and_gate_inst : entity work.and_gate
11      port map (a, b, cout);
12
13    xor_gate_inst : entity work.xor_gate
14      port map (a, b, sum);
15  end architecture;
```



- Instances → statement part
- Two instantiation statements
- Positional association
- Inputs must be readable, outputs writable

Formally the part inside the parentheses of the port map clause is referred to as an "association list". As its name suggests, its purpose is to associate or map an interface signal of an instance with some local signal or expression. This can be done using a positional mapping – like in the example on the previous slide – or using a named mapping.

Port Map

HWMoD
WS24

Struct. Mod.
Introduction
Instances
Port Map
Unused Ports
Generic Map
Components
Architecture
Selection


- Association list ◆

In comparison to positional association, named association increases readability and reduces the chance of errors, especially in designs with many signals or ports. By specifying each connection by name, the code becomes more self-explanatory, and modifications, such as adding, removing, or rearranging ports, can be made without affecting the order of other connections. Named association thus provides better clarity, maintainability, and flexibility, and is generally preferable to the positional variant.

Port Map

HWMoD
WS24

Struct. Mod.
Introduction
Instances
Port Map
Unused Ports
Generic Map
Components
Architecture
Selection


- Association list 
- **Named association** oftentimes preferable over positional association because of better
 - clarity
 - maintainability
 - flexibility
 - robustness (w.r.t. connection bugs)

Syntactically the association list of a port clause looks very similar to an aggregate expression for records. If both positional and named associations are used in the same association list, then all positional associations must appear first. Hence, once a named association is used, the rest of the association list must only use named associations. However, although it is supported by the standard, we strongly advise against mixing association styles when creating instances! Note that we will encounter association lists again, when we talk about function and procedure calls in an upcoming lecture. Here mixed association styles can sometimes be beneficial when dealing with default parameters of a subprogram. The formal part of a named association must refer to a port signal of the entity being instantiated. For port maps the actual part can be an expression containing local signals or constants.

Port Map

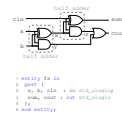
HWMoD
WS24

Struct. Mod.
Introduction
Instances
Port Map
Unused Ports
Generic Map
Components
Architecture
Selection

- Association list 
- **Named association** oftentimes preferable over positional association because of better
 - clarity
 - maintainability
 - flexibility
 - robustness (w.r.t. connection bugs)
- Port map syntax

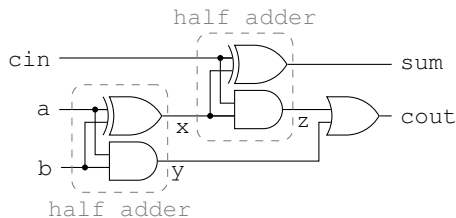

```
port map(association_list)
association_list ::= assoc_elem {, assoc_elem}
assoc_elem ::= [ formal_part => ] actual_part
```

- └ Structural Modeling
 - └ Instances
 - └ **Port Map - Example: Full Adder**



OK, let's look at another code example. You should already know the full adder circuit from a previous lecture. Notice that the full adder can be broken down into two half adders and an "OR"-gate. Hence, we will now implement an architecture that contains two half adder instances.

Port Map - Example: Full Adder



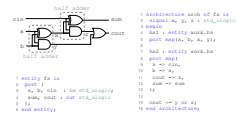
```

1 entity fa is
2   port (
3     a, b, cin : in std_ulogic;
4     sum, cout : out std_ulogic
5   );
6 end entity;
  
```


Structural Modeling

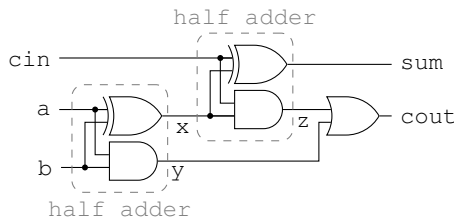
Instances

Port Map - Example: Full Adder



The two instances are named `ha1` and `ha2`. The former one uses positional association in its port map, while the latter uses the named association style. Notice that, although the first port map is much more compact, it is much harder to understand to code without referring back to the half adder entity declaration. For, `ha2` it is immediately clear what the inputs and outputs do. Further notice, how the or-gate producing the final carry-out signal is implemented as a concurrent signal assignment. This shows that instances, concurrent signal assignments as well as processes and other structures can be mixed in one architecture. Also recall that the sequence of statements is irrelevant for the semantics of the code. You might want to pause the video at this point, to really understand the shown circuit and code snippet.

Port Map - Example: Full Adder



```
1 entity fa is
2   port (
3     a, b, cin : in std_ulogic;
4     sum, cout : out std_ulogic
5   );
6 end entity;
```

```
1 architecture arch of fa is
2   signal x, y, z : std_ulogic;
3 begin
4   ha1 : entity work.ha
5     port map(a, b, x, y);
6
7   ha2 : entity work.ha
8     port map(
9       a => cin,
10      b => x,
11      cout => z,
12      sum => sum
13    );
14
15   cout <= y or z;
16 end architecture;
```

Oftentimes it is the case that not all interface signals of an instance are actually used or needed within the instantiating architecture. Reasons for that might be the modular design of components, where the interface is made to be more generic and reusable across different contexts. In such cases, signals may exist to accommodate various configurations, but not all configurations require every signal. Another reason could be that the design includes optional features, controlled by configuration parameters or generics, where only certain signals are used depending on how these options are set. In any case, the question arises how to deal with these situations in code. For that we have to make a distinction between inputs and outputs.

Unused Ports

HWMoD
WS24

Struct. Mod.
Introduction
Instances
Port Map
Unused Ports
Generic Map
Components
Architecture
Selection

Unused outputs

- Use `open` keyword
- Don't leave unconnected!

```

1 [...]
2 signal a, b, c, m : std_ulogic;
3 begin
4   majority : entity work.fa
5     port map (
6       a => a,
7       b => b,
8       c => c,
9       sum => open, -- not connected
10      cout => m
11    );
12 [...]

```

Unused output ports should be clearly marked with the `open` keyword. While not strictly required by the VHDL language specification, this is good practice as it improves readability by making unconnected outputs explicit and helps prevent confusion. It also avoids potential warnings issued by some simulation or synthesis tools. In the example code, we instantiated a full adder but ignored the sum output since as our intent is in using it solely as a majority gate – a function that is implemented by the carry-output.

Unused Ports

HWMoD
WS24

Unused outputs

- Use `open` keyword
- Don't leave unconnected!

```

1 [...]
2 signal a, b, c, m : std_ulogic;
3 begin
4   majority : entity work.fa
5     port map (
6       a => a,
7       b => b,
8       c => c,
9       sum => open, -- not connected
10      cout => m
11    );
12 [...]

```

Unused outputs	Unused inputs
<ul style="list-style-type: none"> Use <code>open</code> keyword Don't leave unconnected! 	<ul style="list-style-type: none"> Connect to constant If not connected → default value
<pre> 1 [...] 2 signal a, b, c, m : std_ulogic; 3 begin 4 majority : entity work.fa 5 port map (6 a => a, 7 b => b, 8 c => c, 9 sum => open, -- not connected 10 cout => m 11); 12 [...] </pre>	<pre> 1 [...] 2 signal a, b, sum, cout : std_ulogic; 3 begin 4 half_adder : entity work.fa 5 port map (6 a => a, 7 b => b, 8 c => '0', -- constant 9 sum => sum, 10 cout => cout 11); 12 [...] </pre>

For unused inputs, we recommend assigning them a constant value, as demonstrated in the example code. It's also possible to use the `open` keyword or simply leave them unconnected. In this case, the port's default value - specified in the entity declaration - will be applied. If no default value is specified, a compilation error will be raised. However, for the same reason as already mentioned for outputs we strongly advise on using constant values.

Unused Ports

HWMoD
WS24

Struct. Mod.
Introduction
Instances
Port Map
Unused Ports
Generic Map
Components
Architecture
Selection

Unused outputs

- Use `open` keyword
- Don't leave unconnected!

```

1 [...]
2 signal a, b, c, m : std_ulogic;
3 begin
4   majority : entity work.fa
5     port map (
6       a => a,
7       b => b,
8       c => c,
9       sum => open, -- not connected
10      cout => m
11    );
12 [...]

```

Unused inputs

- Connect to constant
- If not connected → default value

```

1 [...]
2 signal a, b, sum, cout : std_ulogic;
3 begin
4   half_adder : entity work.fa
5     port map (
6       a => a,
7       b => b,
8       c => '0', -- constant
9       sum => sum,
10      cout => cout
11    );
12 [...]

```

Similar to how the ports of an instance are connected to local signals, there exists an analogous mechanism for generics in the form of the generic-map clause. The generic-map clause also takes an association list and, hence, also supports named and positional association.

Generic Map

HWMoD
WS24

Struct. Mod.
Introduction
Instances
Port Map
Unused Ports
Generic Map
Components
Architecture
Selection

- Generic map syntax
`generic map (association_list)`
- Must appear before port map

Using named association for generics provides the same benefits as for ports, such as improved clarity and reduced potential for errors.

Generic Map

HWMoD
WS24

Struct. Mod.
Introduction
Instances
Port Map
Unused Ports
Generic Map
Components
Architecture
Selection

- Generic map syntax
`generic map (association_list)`
- Must appear before port map
- Use named association and avoid the positional style

A major difference compared to mapping port signals is that the actual part of a generic association must always be a compile-time constant expression. This means it cannot depend on runtime data or dynamically changing values, such as signals. This ensures that the generic values are determined and fixed during simulation and synthesis. Oftentimes generics are mapped to generics of the instantiating architecture which allows to pass parameters down through a design hierarchy

Generic Map

HWMoD
WS24

Struct. Mod.
Introduction
Instances
Port Map
Unused Ports
Generic Map
Components
Architecture
Selection

- Generic map syntax
`generic map (association_list)`
- Must appear before port map
- Use named association and avoid the positional style
- Formal parts must be compile-time constants (generics cannot be connected to signals)

Structural Modeling

Instances

Generic Map – Example: Multiplexer

```

1 entity mux is
2   generic (
3     N : positive
4   );
5   port (
6     c : in std_ulogic;
7     a : in std_ulogic_vector(N-1 downto 0);
8     b : in std_ulogic_vector(N-1 downto 0);
9     o : out std_ulogic_vector(N-1 downto 0)
10  );
11 end mux;
12
13 architecture arch of mux is
14 begin
15   o <= a when c = '0' else b;
16 end architecture;

```



Let's consider this generic two-to-one multiplexer, whose data width can be configured using the generic N . By now, the shown code snippet should be quite clear to you. However, you can pause the video and take some time to understand it.

Generic Map – Example: Multiplexer

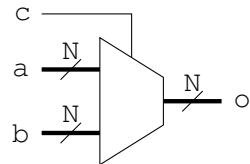
HWMoD
WS24

Struct. Mod.
Introduction
Instances
Port Map
Unused Ports
Generic Map
Components
Architecture
Selection

```

1 entity mux is
2   generic (
3     N : positive
4   );
5   port (
6     c : in std_ulogic;
7     a : in std_ulogic_vector(N-1 downto 0);
8     b : in std_ulogic_vector(N-1 downto 0);
9     o : out std_ulogic_vector(N-1 downto 0)
10  );
11 end entity;
12
13 architecture arch of mux is
14 begin
15   o <= a when c = '0' else b;
16 end architecture;

```



Structural Modeling

Instances

Generic Map – Example: Multiplexer (cont'd)



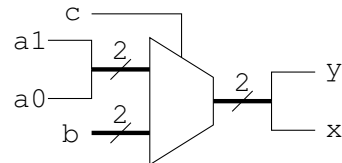
Let's say we want to instantiate this entity in some other architecture with a data width of two.

Generic Map – Example: Multiplexer (cont'd)

```

1  [...]
2  signal c : std_ulogic;
3  signal a0, a1 : std_ulogic;
4  signal b : std_ulogic_vector(1 downto 0);
5  signal x, y : std_ulogic;
6  begin
7    mux_inst : entity work.mux
8      generic map (N => 2)
9      port map (
10       c => c,
11       a => a1 & a0,
12       b => b,
13       o(0) => x,
14       o(1) => y
15     );
16  [...]

```



Structural Modeling

Instances

Generic Map – Example: Multiplexer (cont'd)



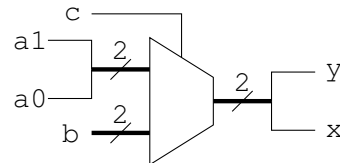
Hence, we use a generic map clause that maps the generic `N` to the constant value two.

Generic Map – Example: Multiplexer (cont'd)

```

1 [...]
2 signal c : std_ulogic;
3 signal a0, a1 : std_ulogic;
4 signal b : std_ulogic_vector(1 downto 0);
5 signal x, y : std_ulogic;
6 begin
7   mux_inst : entity work.mux
8     generic map (N => 2)
9   port map (
10     c => c,
11     a => a1 & a0,
12     b => b,
13     o(0) => x,
14     o(1) => y
15   );
16 [...]

```



Structural Modeling

Instances

Generic Map – Example: Multiplexer (cont'd)



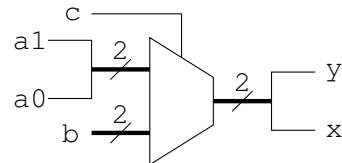
Now, the inputs `a` and `b` as well as the output `o` can be connected to `std_ulogic_vector` signals of length two.

Generic Map – Example: Multiplexer (cont'd)

```

1  [...]
2  signal c : std_ulogic;
3  signal a0, a1 : std_ulogic;
4  signal b : std_ulogic_vector(1 downto 0);
5  signal x, y : std_ulogic;
6  begin
7    mux_inst : entity work.mux
8      generic map (N => 2)
9    port map (
10      c => c,
11      a => a1 & a0,
12      b => b,
13      o(0) => x,
14      o(1) => y
15    );
16  [...]

```



Structural Modeling

Instances

Generic Map – Example: Multiplexer (cont'd)



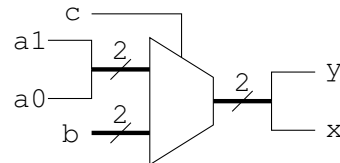
This example snippet also shows the flexibility of association lists – specifically that the formal and actual parts are not restricted to be simple identifiers.

Generic Map – Example: Multiplexer (cont'd)

```

1  [...]
2  signal c : std_ulogic;
3  signal a0, a1 : std_ulogic;
4  signal b : std_ulogic_vector(1 downto 0);
5  signal x, y : std_ulogic;
6  begin
7    mux_inst : entity work.mux
8      generic map (N => 2)
9      port map (
10       c => c,
11       a => a1 & a0,
12       b => b,
13       o(0) => x,
14       o(1) => y
15     );
16  [...]

```



Structural Modeling

Instances

Generic Map – Example: Multiplexer (cont'd)

```

1  [..]
2  signal c : std_ulogic;
3  signal a0, a1 : std_ulogic;
4  signal b : std_ulogic_vector(1 downto 0);
5  signal x, y : std_ulogic;
6
7  begin
8    mux_inst : entity work.mux
9      generic map (N => 2)
10     port map (
11       c => c,
12       a => a1 & a0,
13       b => b,
14       x => x,
15       y => y
16     );
17  end

```



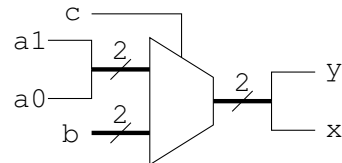
Notice, that the input a is mapped to two single-bit signals a0 and a1 using the concatenation operator.

Generic Map – Example: Multiplexer (cont'd)

```

1  [...]
2  signal c : std_ulogic;
3  signal a0, a1 : std_ulogic;
4  signal b : std_ulogic_vector(1 downto 0);
5  signal x, y : std_ulogic;
6  begin
7    mux_inst : entity work.mux
8      generic map (N => 2)
9      port map (
10       c => c,
11       a => a1 & a0,
12       b => b,
13       o(0) => x,
14       o(1) => y
15     );
16  [...]

```



Structural Modeling

Instances

Generic Map – Example: Multiplexer (cont'd)



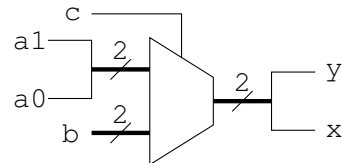
Moreover, the individual bits of output `o` are mapped separately to the local signals `x` and `y`.

Generic Map – Example: Multiplexer (cont'd)

```

1  [...]
2  signal c : std_ulogic;
3  signal a0, a1 : std_ulogic;
4  signal b : std_ulogic_vector(1 downto 0);
5  signal x, y : std_ulogic;
6  begin
7    mux_inst : entity work.mux
8      generic map (N => 2)
9      port map (
10       c => c,
11       a => a1 & a0,
12       b => b,
13       o(0) => x,
14       o(1) => y
15     );
16  [...]

```



You have probably noticed that in all instantiation examples in this lecture, we had to specify more than just the name of the entity. In particular, we always had to explicitly state that we want to instantiate an entity using the `entity` keyword followed by the name of the library the entity resides in – in our case the default library "work" – and the actual name of the entity.

Components

HWMoD
WS24

Struct. Mod.
Introduction
Instances
Components
Architecture
Selection

This was necessary because we did not declare components to our entities. To draw a comparison to the C programming language, you can view a component as the prototype of an entity. The prototype contains only interface information, without providing any details about its internal implementation. Just as function prototypes in C allow you to reference functions before they are defined, component declarations in VHDL allow you to instantiate and reference entities in your design without knowing the full implementation at that point.

Components

HWMoD
WS24

Struct. Mod.
Introduction
Instances
Components
Architecture
Selection

- Components are “*entity prototype*”

As shown in the formal syntax specification, `entity` and `component` declarations look quite similar. However, as we are only interested in the interface and not in any implementation details components do not have a declarative or statement part.

Components

HWMoD
WS24

Struct. Mod.
Introduction
Instances
Components
Architecture
Selection

- Components are "entity prototype"
- Component declaration syntax

```
component NAME is  
[ generic ( {generic_element;} generic_element ); ]  
[ port ( {port_element;} port_element ); ]  
end component;
```

In the C programming language the function prototypes are usually put into header files, while their implementation is kept separate in some C file. In VHDL you can do a similar thing, by putting components in packages and include them elsewhere in you design. However, it is also possible to declare them directly in declarative parts of architectures.

Components

HWMoD
WS24

Struct. Mod.
Introduction
Instances
Components
Architecture
Selection

- Components are "*entity prototype*"
- Component declaration syntax


```
component NAME is
[ generic ( {generic_element;} generic_element ); ]
[ port ( {port_element;} port_element ); ]
end component;
```
- Components can be put in
 - packages
 - declarative part of architectures (blocks and generate statements)

Now the question may arise, why even bother with components? As shown in the code examples the component instantiation syntax is a little simpler and more compact. Generally, the same arguments as for structuring C programs into header and source files apply here. Components in VHDL, like functions and modules in C, help break down complex designs into smaller, more manageable parts, promoting modularity and reusability and a separation of concerns. Moreover, they help to relax constraints on the compilation order and can, hence, improve compilation and synthesis speeds. Sometimes, it can also be the case that you don't have access to a modules' implementation – neither its entity nor its architecture. This can, for example, occur when you're working with precompiled IP cores or third-party libraries provided by FPGA vendors or other companies. In such cases, you are given a black box view of the module, where the internal workings are hidden, but the external interface is available in the form of a component declaration. You can again compare this to the situation in software development where you have a precompiled library, without access to its source code. In both cases, you are provided with the interface – in software, typically in the form of a header file – that describes the inputs and outputs, or functions and their arguments. Components are also needed when you are working with mixed-language designs – for example when you mix VHDL and Verilog in a single project.

Components (cont'd)

HWMoD
WS24

Struct. Mod.
Introduction
Instances
Components
Architecture
Selection

- **Instantiation example**
 - Entity: `i : entity work.fa port map (...);`
 - Component: `i : fa port map (...);`
 - Component (explicit): `i : component fa port map (...);`
- **Modularity, abstraction and separation of concerns**
- **Compilation order**
- **Entity may not always be available**
- **Mixed-language designs**

As we have discussed in a previous lecture, an entity can have multiple different architectures. So you might ask yourself, how we can select a particular one when creating an instance of a module. VHDL offers two possibilities to do that.

Architecture Selection

HWMoD
WS24

Struct. Mod.
Introduction
Instances
Components
Architecture
Selection
Entity Instantiations
Configurations

- Multiple different architectures possible for a single entity
 - ⇒ How to select one?
- Two possibilities
 - Specify architecture, when an **entity** is instantiated
 - Caution:** This does not work for components!
 - Configurations

```

1 entity mystery is
2   port (
3     a, b : in std_ulogic;
4     x : out std_ulogic);
5   --
6   --
7   --
8   --
9   --
10  --
11  --
12  --
13  --
14  --
15  --
16  --
17  --
18  --
19  --
20  --
21  --
22  --
23  --
24  --
25  --
26  --
27  --
28  --
29  --
30  --
31  --
32  --
33  --
34  --
35  --
36  --
37  --
38  --
39  --
40  --
41  --
42  --
43  --
44  --
45  --
46  --
47  --
48  --
49  --
50  --
51  --
52  --
53  --
54  --
55  --
56  --
57  --
58  --
59  --
60  --
61  --
62  --
63  --
64  --
65  --
66  --
67  --
68  --
69  --
70  --
71  --
72  --
73  --
74  --
75  --
76  --
77  --
78  --
79  --
80  --
81  --
82  --
83  --
84  --
85  --
86  --
87  --
88  --
89  --
90  --
91  --
92  --
93  --
94  --
95  --
96  --
97  --
98  --
99  --
100 --

```

When creating an instance of an entity, it is possible to specify the name of the architecture that should be used after the entity name in parentheses. Note however, that this only works for entity instantiation and not for components. In the example code on the left we have declared a simple entity with two inputs and one output called "mystery". It has two different architectures, a1 and a2, implementing the behavior of an "XOR" and an "and"-gate, respectively.

Entity Instantiations

HWMoD
WS24

Struct. Mod.
Introduction
Instances
Components
Architecture
Selection
Entity Instantiations
Configurations

```

1 entity mystery is
2   port (
3     a, b : in std_ulogic;
4     x : out std_ulogic
5   );
6 end entity;
7
8 architecture a1 of mystery is
9 begin
10  x <= a and b;
11 end architecture;
12
13 architecture a2 of mystery is
14 begin
15  x <= a xor b;
16 end architecture;

```

Structural Modeling

Architecture Selection

Entity Instantiations

```

1 entity mystery is
2   port (
3     a, b : in std_ulogic;
4     x : out std_ulogic);
5   is
6   end entity;
7
8   -- component a1 of mystery is
9   --   port map (a, b, x);
10  -- end architecture;
11
12  -- component a2 of mystery is
13  --   port map (a, b, x);
14  -- end architecture;
15
16  -- component work of ha is
17  --   and_gate : entity work.mystery(a1)
18  --     port map (a, b, cout);
19  --   xor_gate : entity work.mystery(a2)
20  --     port map (a, b, sum);
21  -- end architecture;

```

On the right side we can now see how to use this gate to implement the half adder circuit, we have already seen on a previous slide. Notice how the architecture names appear next to name of the entity names.

Entity Instantiations

HWMoD
WS24

Struct. Mod.
Introduction
Instances
Components
Architecture
Selection
Entity Instantiations
Configurations

```

1 entity mystery is
2   port (
3     a, b : in std_ulogic;
4     x : out std_ulogic
5   );
6 end entity;
7
8 architecture a1 of mystery is
9 begin
10  x <= a and b;
11 end architecture;
12
13 architecture a2 of mystery is
14 begin
15  x <= a xor b;
16 end architecture;

```

```

1 entity ha is
2   port (
3     a, b : in std_ulogic;
4     sum, cout : out std_ulogic
5   );
6 end entity;
7
8 architecture arch of ha is
9 begin
10  and_gate : entity work.mystery(a1)
11    port map (a, b, cout);
12
13  xor_gate : entity work.mystery(a2)
14    port map (a, b, sum);
15 end architecture;

```



The other possibility to select between architectures is via the use of an VHDL language feature called configurations. However, we don't want to go into too much detail about this feature as its syntax is a little involved. It suffices, if you know that a configuration, can – among other things – select a particular architecture for a particular instance in a design, without the need to change the architecture in which this instance is created. Let us close this lecture with a simple practical example for a configuration, that we will also encounter in the exercise part of this course. For tasks that must be loaded onto the FPGA board you will have a top-level entity named debug-top, which contains two instances – the instance of the user-top design and the debug-core, which provides some debugging capabilities. The figure on the right illustrates this design hierarchy. The user-top design has a fixed entity specification, and you will have to implement different architectures for the various tasks of the exercise sheet. Since we don't want to always change the debug-top architecture when we want to switch between different architectures for the individual tasks we can use a configuration. The code snippet on the left of the slide shows how this can look like. It creates a configuration named debug-top-conf for the debug-top entity, which has a single architecture named debug-top-arch. For this architecture it then specifies that the instance named user-top-inst of the user-top entity shall use the arch-A architecture. Now, to change the architecture we only have to change line 4 of our configuration.

Configurations

26

HWMoD
WS24

Struct. Mod.
Introduction
Instances
Components
Architecture
Selection
Entity Instantiations
Configurations

```

1 configuration dbg_top_conf of dbg_top is
2   for dbg_top_arch
3     for user_top_inst : user_top
4       use entity user_top(arch_A);
5     end for;
6   end for;
7 end configuration;

```

dbg.top

dbg_core_inst : dbg_core

debugging logic for
the user top design

user_top_inst : user_top

multiple architectures
arch_A, ..., arch_Z

Thank you for listening! We recommend you to immediately take the self-check test in TUWEL, to see if you understood the material presented in this lecture.

HWMMod
WS24

Struct. Mod.

Introduction

Instances

Components

Architecture

Selection

Entity Instantiations

Configurations

Lecture Complete!