

Now that we have some basic knowledge about the programming language VHDL, we will take a look at how it is actually used to describe hardware.



Ent. & Arch. Introduction Entities Architectures

#### Hardware Modeling [VU] (191.011) - WS24 -

Describing Hardware – Entities and Architectures

Florian Huemer & Sebastian Wiedemann & Dylan Baumann

WS 2024/25

Modified: 2025-03-12, 16:28 (21636bb)

# —Describing Hardware – Entities and Architectures Introduction Introduction

	VHDL Code Synthesiznable VHDL Code
Recall	

Recall that, only a subset of the language features in VHDL is synthesizable. This means that only this subset can be used to actually describe hardware. Nevertheless, the non-synthesizable parts of the language are vitally important to, for example, verify the behavior of a design in a testbench. So far we have treated VHDL as just another, maybe a little exotic, programming language. Now, we want to focus on the circuit-level semantics of its language constructs. At the end of the lecture, we will have implemented our first combinational digital circuit.



Interface specification of a module
 Interface signals
 Parameters
 Multiple different architectures possible

When we are designing a circuit, one of the key tasks is to define its interface to the outside world. As already introduced in a previous lecture, in VHDL this is done using the entity structure. It defines the inputs, outputs, and configuration parameters of a module. However, it does not specify how the module behaves internally – that is handled by its architectures. Note that a single entity can have multiple different architectures.

# Extraction Interface specification of a module Interface signals Interface signals Interface signals Parameters Interface different architectures possible

Interface specification of a module
 Interface specification
 Margine different architectures possible
 Marging different architectures possible
 Social and other
 Social and other

To draw a comparison to other programming languages, the entity-architecture concept can be compared to abstract classes or interfaces and their implementations. However, unlike interfaces in for example Java or C# an architecture can only have a single entity. Another analogy closer to hardware – that fits slightly better – is the picture of a microchip and its socket. The socket represents the interface, while the actual microchip can be changed – for example different compatible processors with varying performance.



Eftly declaration syntax entity NAUME is [generic\_element;] generic\_element ); ] [pott ([got\_element;] pot\_element ); ] [entity\_declarative\_part] end entity\_statement\_part ] end entity;

Entities are declared using the entity keyword, followed by an identifier representing its name and the keyword is. Then four blocks referred to as the generics and ports clause and the entity declarative and statement parts are listed. Note that, as explicitly marked in the syntax specification, each of these blocks is optional. We have already seen this in previous code examples, where only completely empty entities have been used so far.

#### Entities (cont'd)

#### HWMod WS24

Ent. & Arch. Introduction Entities Ports Generics Unconstrained Types Architectures

```
20
```

#### Entity declaration syntax

```
entity NAME is
  [ generic ( {generic_element; } generic_element ); ]
  [ port ( {port_element; } port_element ); ]
  [ entity_declarative_part ]
  [ begin entity_statement_part ]
end entity;
```

The generic clause is introduced with the generic keyword, which is followed by a semicolon-separated list of generic elements wrapped in parentheses. Note that 2019 the last element in the list is not terminated with a semicolon. Having an additional semicolon at this position is only allowed since VHDL-2019 – in previous versions this syntax leads to a compilation error. Generics can be used to define parameters that customize the behavior or structure of a module, improving code reuse and flexibility. We will go into further depth on generics on a following slide.

# Entities (cont'd) FWMod WS2 For & Arch. For service For Ser

 Enthy declaration syntax entity loads: is (propertia\_liberati) preserie\_sliment ); ] [ prot( [processionseri) preserie\_sliment ); ] [ entity\_declarative\_alernation [ entity\_declarative\_pression entity\_declarative\_pressio

The port clause is marked with the port keyword and is then followed by a list of port elements using the same syntax as the generic block. As for the generics, the last element of the list is "NOT" followed by a semicolon. The signals defined in and entity's port list define the actual physical interface of the module to the outside world. If you recall the Java and C# analogies from before, you can view the ports as the methods of an abstract class, and the generics as the parameters of the constructor.

#### Entities (cont'd)

#### HWMod WS24

Ent. & Arch. Introduction Entities Ports Generics Unconstrained Types

# 20

#### Entity declaration syntax

entity NAME is

```
[ generic_element; } generic_element ); ]
```

```
[ port ( {port_element; } port_element ); ]
```

```
[ entity_declarative_part ]
```

```
[ begin entity_statement_part ]
```

end entity;

- Generics: configuration/parameters
- Ports: physical I/O signals

Ently declaration syntax
 entity declaration syntax
 entity (declaration syntax)
 generatic (light=limit()) generatic\_slamment);
 [ entity\_declarative\_part]
 entity\_declarative\_part]
 entity\_declarative\_part
 end entity;
 end entiy;
 end entity;

As the name suggest the declarative parts simply contains a range of declarations for – among other things – constants, types, or subprograms. The objects declared here are visible only to the architectures implementing the respective entity.

#### Entities (cont'd)

#### HWMod WS24

Ent. & Arch. Introduction Entities Ports Generics Unconstrained Types

# 20

#### Entity declaration syntax

#### entity NAME is

```
[ generic ( {generic_element; } generic_element ); ]
```

```
[ port ( {port_element; } port_element ); ]
```

```
[ entity_declarative_part ]
```

```
[ begin entity_statement_part ]
```

```
end entity;
```

- Generics: configuration/parameters
- Ports: physical I/O signals
- Declarations: constants, types, subprograms, etc.

Entry declaration systax
 entry declaration systax
 entry documents // generation/constrainty // generation/constrainty // generation/constrainty // [
 entry/constraints/systamming\_entry]
 end entry/
 end entry/
 end entry/
 bookstrainty approximation
 bookstrainty approximation
 bookstrainty approximation
 bookstrainty approximation
 bookstrainty approximation
 bookstrainty approximation
 bookstrainty
 bookstrainty

Finally, the begin keyword introduces the statement part, which can for instance be used to implement constraints or static sanity checks on the generic parameters.

#### Entities (cont'd)

#### HWMod WS24

Ent. & Arch. Introduction Entities Ports Generics Unconstrained Types

### 20

#### Entity declaration syntax

#### entity NAME is

```
[ generic_element; } generic_element ); ]
```

```
[ port ( {port_element; } port_element ); ]
```

```
[ entity_declarative_part ]
```

```
[ begin entity_statement_part ]
```

end entity;

- Generics: configuration/parameters
- Ports: physical I/O signals
- Declarations: constants, types, subprograms, etc.
- Statements: parameter checks

# —Describing Hardware – Entities and Architectures —Entities —Ports

OK, let's start with taking a closer look at the ports section. The declaration of port elements looks very similar to the declaration of variables or constants we have already seen in previous lectures. Each port element starts with a name, followed by a colon and the optional mode specifier. Then the datatype of the port is specified. Finally, an optional default value can be added. The semantics of this will be explained in an upcoming lecture. Note that similar to variable or constant declarations the port name can also be a comma-separated list of identifiers. The semantics of this syntax should be obvious.

# FWX0d WSd2 Image: Description of the content of th

Det element declaration syntax port\_slement ::\* PORT\_JUARE : [mode] DATA\_TYPE [:= default\_value] # Mode Possible Values: in, out, inout, buffer, linkage Default (finode is omthod) in Default direction of a signal (can it be read, within or both)

The mode of a port basically specifies its direction. The VHDL standard defines five possible modes, with "in" being the default one. However, in this course we will only use the first three – that is in, out and inout. The other ones are rarely used and will therefore not be covered in this course. As you might be able to deduce from the mode names, "in" and "out" define circuit inputs and outputs, respectively. The mode "inout" is only used for interface signals that are actively driven by the design as well as read back. This is for example required to implement open-drain outputs or bus protocols where it is required to disconnect from a shared bus wire because other devices actively drive it. However, for now we will just use the modes "in" and "out". The "inout" mode will be covered in more detail in an upcoming lecture. In the context of synthesizable entities the port datatype must of course be a synthesizable datatype, such as boolean, bit or integer. Although it seems very natural to represent the logical binary state of a signal wire using a Boolean value, in practice these datatypes are rarely used for interface signals. As will be shown in an upcoming lecture, even in a digital system it is beneficial to model more signal states than just high and low. An example where the boolean values are insufficient is the shared bus wire we just mentioned. However, until we learn about more specialized datatypes, we will stick to Booleans.

	Ports
HWMod WS24	Port element declaration syntax port_element ::=
Ent. & Arch. Introduction Entities Ports Generics Unconstrained Types Architectures	<pre>PORT_NAME : [mode] DATA_TYPE [:= default_value] Mode Possible Values: in, out, inout, buffer, linkage Default (if mode is omitted): in Defines the direction of a signal (can it be read, written or both)</pre>

# —Describing Hardware – Entities and Architectures —Entities —Ports



The example entity AND-gate on this slide, has two inputs – named a and b – and one output x. As its name suggests this entity models the interface of a 2-input "and" gate. We will add an appropriate architecture on an upcoming slide.

#### Ports HWMod Port element declaration syntax **WS24** port element ::= PORT NAME : [mode] DATA TYPE [:= default value] Mode Ports Possible Values: in, out, inout, buffer, linkage Default (if mode is omitted): in Defines the direction of a signal (can it be read, written or both) Example entity 1 entity and\_gate is and\_gate 2 port ( 3 a : in boolean; а b : in boolean; 4 → X 5 x : out boolean -- no semicolon here! b 6 ); 7 end entity;

With ports we can specify the physical interface signals of a module. However, when designing hardware oftentimes we also want to configure other design parameters that are not actual physical inputs. Consider the example of a RAM module, an important and widely used memory element in digital design. A typical design might use many RAMs with different address and data widths. It would be quite a tedious task to explicitly design RAM modules for all the different configurations one might need in a design. Moreover, an approach like this would result in a hard-to-maintain code. It would be much better to define one RAM module with parameters that select the data and address widths. This is where the generics come into play.

default_value]

# —Describing Hardware – Entities and Architectures —Entities —Generics

# Generic element declaration syntax generic\_element ::= GENERIC\_NAME : DATA\_TYPE [:= default\_value] # No mode required (all generics are "constant inputs")

As already mentioned, generics are used to parametrize entities, allowing to create modules that can be adapted to different use cases with different settings, such as sizes, delays, or other operational parameters. The declaration of generics looks very similar to a port declaration – the only difference is that no mode is needed as all generics can be viewed as constant inputs. Generics will become important when we talk about the instantiation of modules in the next lecture.

# FMMod VS24 • Generic element declaration syntax generic\_element ::= GENERIC\_NAME : DATA\_TYPE [:= default\_value] • No mode required (all generics are "constant inputs")

 Generic element declaration syntax generic\_slement ::= cENERIC\_SIMME : DATA\_TYPE [:= default\_value]
 No mode required (all generics are "constant inputs")
 Basically constants in the architecture

From the point of view of an architecture implementing an entity, generics can simply be viewed as constants, that are fixed at compile-time. Hence, even in synthesizable modules, it is possible to use non-synthesizable datatypes here. An example would be a time constant for the clock period or other timing parameters.

#### Generics

#### HWMod WS24

Ent. & Arch. Introduction Entities Ports Generics Unconstrained Types Architectures

- Generic element declaration syntax
  generic\_element ::=
  GENERIC\_NAME : DATA\_TYPE [:= default\_value]
- No mode required (all generics are "constant inputs")
- Basically constants in the architecture



Note that port declarations may refer to generics and use them to constrain port datatypes. This is also done in the example code shown on this slide. The entity is quite similar to the one on the previous slide. However, the two input signals have been replaced by an input vector named "i". The width of this vector is defined by the generic "N", allowing you to create AND-gates with arbitrary many inputs based on a single entity. This is a code pattern, you will encounter quite often in VHDL code.

	Generics
HWMod WS24 Ent. & Arch. Introduction Entities Ports Genetics Unconstrained Types Architectures	<ul> <li>Generic element declaration syntax generic_element ::= GENERIC_NAME : DATA_TYPE [:= default_value]</li> <li>No mode required (all generics are "constant inputs")</li> <li>Basically constants in the architecture</li> <li>Example entity</li> </ul>
	<pre>1 entity and_gate is 2 generic ( 3 N : integer := 2 4 ); 5 port ( 6 i : in boolean_vector(N-1 downto 0); 7 x : out boolean 8 ); 9 end entity;</pre> and_gate $x = \frac{N}{N}$

# —Describing Hardware – Entities and Architectures —Entities —Unconstrained Types

Unconstrained generic example

i entry ope is

service and a service an

Note that it is also possible to use unconstrained types for generics and ports. For certain datatypes like strings, this is even necessary to make reasonable use of them as generics because not all possible values might have the same character length. In fact, in practice strings are sometimes used to enable or disable certain settings in an entity. The first example code shows how something like this can look like. Note how the two strings "yes" and "no", that the entity expects the generic to have, are of different length.

#### Unconstrained Types

Unconstrained generic example

#### HWMod WS24

Ent. & Arch. Introduction Entities Ports Generics Unconstrained Types Architectures

```
1 entity cpu is
2 generic (
3 ENABLE_BRANCH_PREDICTION : string := "YES"; -- "NO"
4 [...]
5 );
6 [...]
7 end entity;
```

# —Describing Hardware – Entities and Architectures —Entities —Unconstrained Types

Unconstrained datatypes can also come in handy for port declarations. The second example shows yet another variant of the AND-gate entity. This time no generic is used and the input i is simply left unconstrained. Whenever this entity is instantiated, the width of this input signal is then implicitly constrained by the vector signal that is connected to i.

#### **Unconstrained Types**

#### HWMod WS24

Introduction Entitles Ports Generics Unconstrained Types Architectures

# 1 entity cpu is 2 generic ( 3 ENABLE\_BRANCH\_PREDICTION : string := "YES"; -- "NO" 4 [...]

5 ); 6 [...] 7 end entity;

#### Unconstrained port example

Unconstrained generic example

```
1 entity and_gate is
2  port (
3         i : in boolean_vector;
4         x : out boolean
5    );
6 end entity;
```

# Describing Hardware – Entities and Architectures Architectures Architectures

Now that we are familiar with how to specify module interfaces in VHDL, let's turn to architectures and see how they can be used to describe actual digital circuits. The VHDL standard defines architectures as the body of a design entity.

#### Architectures

HWMod WS24

Ent. & Arch. Introduction Entities Architectures Concurrent Assignments Signal Declaratio ■ VHDL standard ♦ :

"An architecture body defines the body of a design entity."

# —Describing Hardware – Entities and Architectures —Architectures —Architectures

 VHDL standard \$: "An architecture body defines the body of a design entity."
 Contains the actual circuit description

Thus, an architecture defines the internal behavior and structure of a digital design, and essentially specifies how the inputs of an entity have to be processed to create the output signals.

#### Architectures

HWMod WS24

Ent. & Arch. Introduction Entities Architectures Concurrent Assignments Signal Declaratio "An architecture body defines the body of a design entity."

Contains the actual circuit description

# Describing Hardware – Entities and Architectures Architectures Architectures

An architecture is declared using the architecture keyword followed by an identifier, representing its name, the keyword of and another identifier specifying the entity this architecture belongs to.

#### Architectures

HWMod WS24

Ent. & Arch. Introduction Entities Architectures Concurrent Assignments Signal Declaration

#### VHDL standard 🔶 :

"An architecture body defines the body of a design entity."

- Contains the actual circuit description
- Architecture declaration syntax

```
architecture NAME of ENTITY_NAME is
    architecture_declarative_part
begin
    architecture_statement_part
```

end architecture;

# —Describing Hardware – Entities and Architectures —Architectures —Architectures



After the keyword is follow the architecture's declarative and statement part, which are separated by the begin keyword. As you can see from the syntax specification, both of these parts can contain zero or more elements. The VHDL standard refers to these elements as block declarative items and concurrent statements, respectively.

#### **Architectures** HWMod **WS24** "An architecture body defines the body of a design entity." Contains the actual circuit description Architecture declaration syntax Architectures architecture NAME of ENTITY NAME is architecture declarative part begin architecture\_statement\_part end architecture; Declarative/statement part architecture\_declarative\_part ::= { block\_declarative\_item } architecture\_statement\_part ::= { concurrent\_statement }

# —Describing Hardware – Entities and Architectures —Architectures —Architecture (cont'd)

 Declarative part (block declarative items) Signals Constants Types Sub-programs enn

As the name suggests, the declarative part, contains declarations for various VHDL language constructs, such as constants, signals, types, subprograms and others.

#### Architecture (cont'd)

#### HWMod WS24

Ent. & Arch. Introduction Entities Architectures Concurrent Assignments Signal Declaratio

- Declarative part (block declarative items) ♦
  - Signals
  - Constants
  - Types
  - Sub-programs
  - etc.

# —Describing Hardware – Entities and Architectures —Architectures —Architecture (cont'd)

 Declarative part (block declarative items) Signals Constants Constants Solutions Solutions Concurrent span assignments Poccesses Instantalisto statements Boldements Boldements Solutions Solutions

There are various types of concurrent statements that can go into the statements part of an architecture. In this lecture we will focus on the concurrent signal assignments. While we have already seen quite a few processes during the previous lectures, as most of the code samples were build around them, we have only used them in non-synthesizable contexts so far. We will discuss processes for synthesizable code, instantiations of entities as well as block and generate statements in upcoming lectures.



# Describing Hardware – Entities and Architectures Architectures Concurrent Signal Assignments – Example: AND gate

sectory mod\_gence is 2 post ( 3 a in honlowsy b in honlowsy b in honlows) 4 yi 7 ead entry; 9 available 9 ava

We will now introduce concurrent signal assignments. Let's start with a very basic example and implement an architecture for the "and"-gate entity we showed before using a concurrent signal assignment. The assignment is located in line 11 of the sample code. Notice that VHDL uses a different assignment operator for signals than for variables.

#### Concurrent Signal Assignments – Example: AND gate

HWMod WS24

Int. & Arch. Introduction Entities Architectures Concurrent Assignments Signal Declaration

port ( 2 3 a : in boolean; b : in boolean; 4 5 x : out boolean 6 ); 7 end entity; 8 9 architecture arch of and\_gate is 10 begin  $x \leq a$  and b; 12 end architecture;

1 entity and\_gate is



# Describing Hardware – Entities and Architectures Architectures Concurrent Signal Assignments – Example: Multiplexer



Let's look at a slightly more complex example, that of a four-to-one multiplexer. As you should already know, the function of a multiplexer is to relay one of the inputs to a single output based on the value of a control input. In our case here, we want to multiplex between four input signals i(0) to i3, for which we need to control inputs c(0) and c(1). To implement the multiplexer we use a so-called conditional signal assignment. Essentially, this is a compact way to specify an if-else-if construct in a single expression. You can compare it to a nested ternary conditional operator in other programming languages such as Java or C. If both control signals are false then the output gets assigned the input i(0). If only c(0) is true, i(1) is selected, and so on. The right side of the slide depicts the circuit symbol that we will use for multiplexers throughout this course.



# Describing Hardware – Entities and Architectures Architectures Signal Declarations

Signal declaration syntax signal NAME : DATA\_TYPE [ := default\_value ] ;

It is of course also possible to declare local signals in an architecture that are not visible to the outside world. As you might have suspected, such declaration go into the declarative part of the architecture. The syntax for that is quite straight-forward and basically looks very similar to a variable or constant declaration that we have already seen multiple times. However, before we further elaborate, it is important to discuss a detail that we more or less glossed over so far.

# Signal Declarations HWMod Ws24 Ent. & Arch. Live Archetecting Scale Declaration syntax Signal NAME : DATA\_TYPE [ := default\_value ] ;

# Describing Hardware – Entities and Architectures Architectures Signal Declarations

In VHDL there is a fundamental difference between objects of type signal – that also includes the ports declared in an entity – and variables. Signals represent and model physical connections between hardware components such as gates or registers. They can be declared in the declarative part of architectures or entities as well as in packages. Furthermore, some architecture statements such as blocks or generate statements can have their own local signals. However, they cannot be declared in subprograms or processes. Here only variable declarations are allowed, representing local values that are only relevant during the evaluation of a process or a subprogram. The major difference comes from the way these two classes of objects are treated during simulation and synthesis. In strongly simplified terms: Signal assignment affect the internal event queue the simulator has to maintain, whereas variable assignments do not and are executed immediately. As we have already seen VHDL even goes so far as to use unique assignment operators for signals and variables, to also highlight this difference syntactically. However, we don't want to go into too much detail here – as this is the content of another lecture.



# Describing Hardware – Entities and Architectures Architectures Example - Full Adder



> point {
3 a, b, oin i in boolean
4 sum, cout i out boolean
5 };

Let's look at a slightly more complex example that uses local signal declarations. A full adder, such as the one shown on the left side of the slide, is a digital circuit with three inputs -a, b, and cin - and two outputs -sum and cout. Its purpose is to add the three single-bit input signals and produce a 2-bit result consisting of the sum bit and a carry output. In order to add multi-bit signals, multiple full adders can be connected together in the form of a ripple-carry adder.

#### Example - Full Adder

#### HWMod WS24

Ent. & Arch. Introduction Entities Architectures Concurrent Assignments Signal Declarations



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

# Describing Hardware – Entities and Architectures Architectures Example - Full Adder



1 satisfy fait 2 part ( 3 part ( 4 same, door, is is bolisson 4 same, door, is fablaban 9 ) 9 ) 9 architecture acts of fais 4 (spale a, y, is bolisson 9 (spale a, spale b) 1 s <- a same by 1 s <- a same b

To implement the circuit the example uses three intermediate signals named "x", "y" and "z". You can see which circuit nodes these signals represent in the schematic on the left. The five concurrent signal assignments correspond to the five gates in the circuit and should be quite straight-forward to understand. However, you can pause the video to convince yourself that this is indeed the case.

#### Example - Full Adder

#### HWMod WS24

Ent. & Arch. Introduction Entities Architectures Concurrent Assignments Signal Declarations



1	entity fa is
2	port (
3	a, b, cin : in boolean;
4	<pre>sum, cout : out boolean</pre>
5	);
6	end entity;
7	
8	architecture arch of fa is
9	<pre>signal x, y, z : boolean;</pre>
10	begin
11	x <= a xor b;
12	y <= a and b;
13	<pre>sum &lt;= cin xor x;</pre>
14	z <= cin and x;
15	cout <= y or z;
16	end architecture;



One key take-away from this code is that the sequence of the assignment statements does not matter in any way. No matter how you arrange the statements in an architecture the semantic of the architecture stays the same. Unlike in software, where statements are executed strictly in sequence, in a circuit a lot of things happen in parallel. The individual gates in a circuit always evaluate their inputs and produce a new output signal level. Hence, you can think of the individual statements in an architecture as individual threads that are all executed in parallel. Hence, we call this "concurrent" signal assignments. This is a key insight that will be very important throughout this course and your journey with VHDL.

#### Example - Full Adder

#### HWMod WS24

Introduction Entitles Architectures Concurrent Assignments Signal Declarations



#### Important

The sequence of the statements in an architecture has no semantic meaning!

1	entity fa is
2	port (
3	a, b, cin : in boolean;
4	sum, cout : out boolean
5	);
6	end entity;
7	
8	architecture arch of fa is
9	<pre>signal x, y, z : boolean;</pre>
10	begin
11	x <= a xor b;
12	y <= a and b;
13	sum <= cin xor x;
14	z <= cin and x;
15	cout <= y or z;
16	end architecture;

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.



Ent. & Arch. Introduction Entities Architectures Concurrent Assignments Signal Declarations

# Lecture Complete!

Modified: 2025-03-12, 16:28 (21636bb)