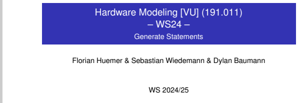
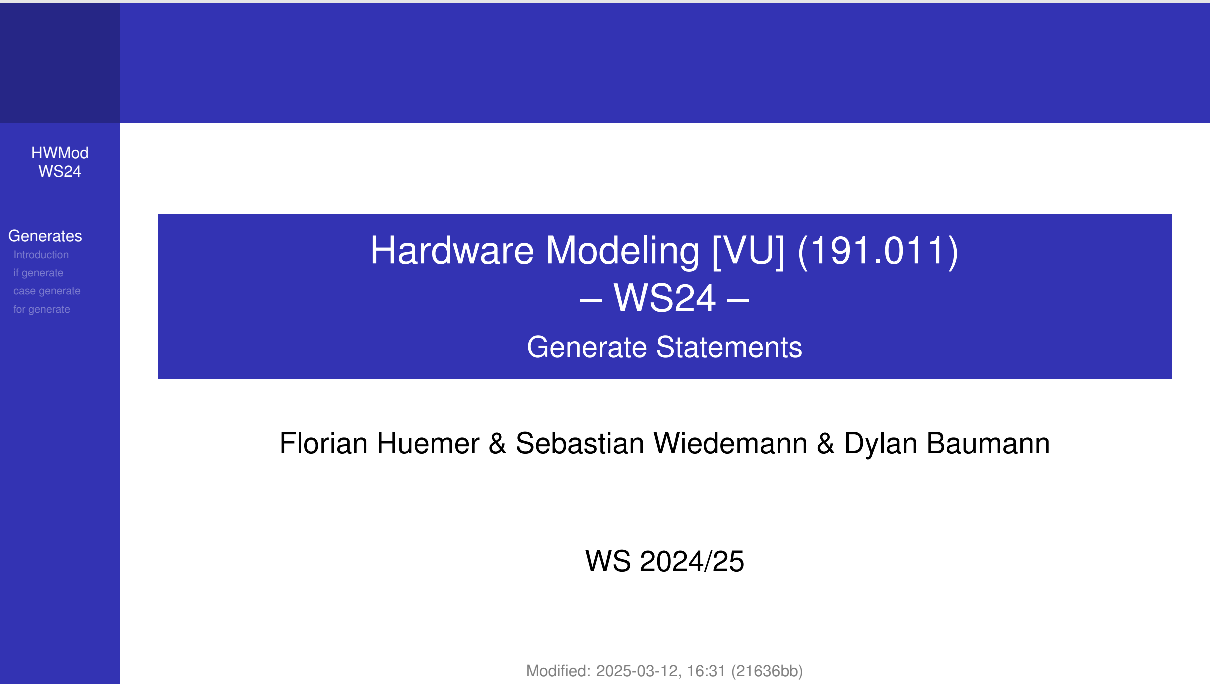


Hardware Modeling [VU] (191.011)
– WS24 –
Generate Statements

Florian Huemer & Sebastian Wiedemann & Dylan Baumann

WS 2024/25

In this lecture, we will explore another essential class of concurrent statements: the generate statements. They are a vital tool in creating flexible and parameterized code structures, allowing designers to efficiently replicate or conditionally instantiate components based on specific parameters.



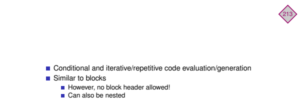
HWMod
WS24

# Hardware Modeling [VU] (191.011) – WS24 – Generate Statements

Florian Huemer & Sebastian Wiedemann & Dylan Baumann

WS 2024/25

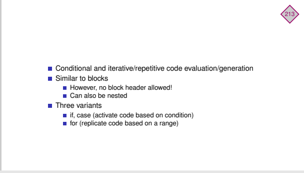
Modified: 2025-03-12, 16:31 (21636bb)


- Conditional and iterative/repetitive code evaluation/generation
- Similar to blocks
  - However, no block header allowed!
  - Can also be nested

Generate statements – or just generates in short – have some similarities to blocks covered in a previous lecture, as they also serve as containers for concurrent statements. However, instead of simply encapsulating the statements, generates can conditionally activate them at compile time or replicate them in a loop-like structure. In combination with generics, generates are a key ingredient for designing flexible and reusable hardware components. Although generates have some syntactical and conceptional relation to blocks, they don't feature block headers. However, similar to blocks, generate statements can also be nested.

## Introduction

- Conditional and iterative/repetitive code evaluation/generation
- Similar to blocks
  - However, no block header allowed!
  - Can also be nested

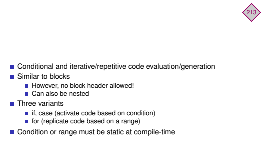

213

- Conditional and iterative/repetitive code evaluation/generation
- Similar to blocks
  - However, no block header allowed!
  - Can also be nested
- Three variants
  - if, case (activate code based on condition)
  - for (replicate code based on a range)

VHDL supports three types of generate statements. If- and case-generates can be used to activate or deactivate certain blocks of code based on one or more conditions. The for-generate statement, on the other hand, is used to replicate a block of code multiple times, similar to a loop. This type of generate statement is especially useful for creating regular structures, such as arrays of components or repeated logic, by iterating over a specified range.

## Introduction

- Conditional and iterative/repetitive code evaluation/generation
- Similar to blocks
  - However, no block header allowed!
  - Can also be nested
- Three variants
  - if, case (activate code based on condition)
  - for (replicate code based on a range)

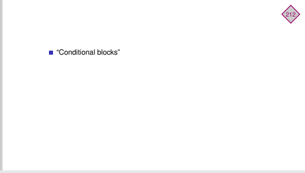

- Conditional and iterative/repetitive code evaluation/generation
- Similar to blocks
  - However, no block header allowed!
  - Can also be nested
- Three variants
  - if, case (activate code based on condition)
  - for (replicate code based on a range)
- Condition or range must be static at compile-time

Note that since we are talking about compile-time evaluated conditions and loops, the condition for if and case-generates, as well as the loop range for for-generates must be static at compile time.

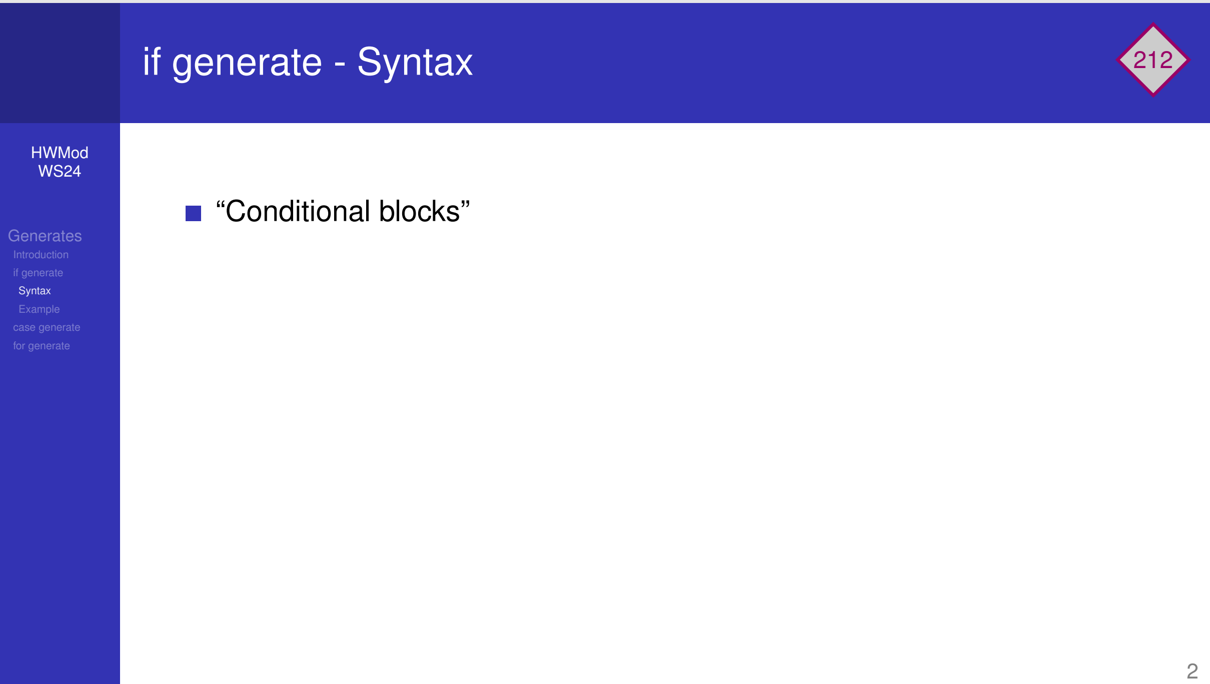
## Introduction

- Conditional and iterative/repetitive code evaluation/generation
- Similar to blocks
  - However, no block header allowed!
  - Can also be nested
- Three variants
  - if, case (activate code based on condition)
  - for (replicate code based on a range)
- Condition or range must be static at compile-time

Generate Statements
  if generate
    **if generate - Syntax**

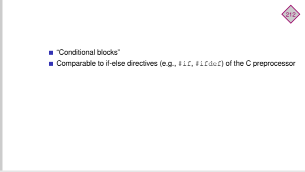


Let’s start by taking a closer look at if-generate statements. If-generates can be viewed as conditional blocks, that is, blocks whose content is only active if a specific condition is true.



## if generate - Syntax

- “Conditional blocks”

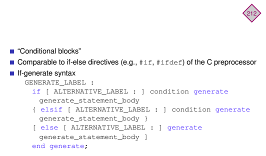


- "Conditional blocks"
- Comparable to if-else directives (e.g., `#if`, `#ifdef`) of the C preprocessor

You can compare if-generates loosely to if-else preprocessor directives in the C programming language. Those directives are evaluated right before the actual compilation and can, thus, be used to instruct the compiler to ignore certain parts of a source file. However, one big difference is that VHDL does not have a preprocessor and the compiler itself evaluates the respective conditions.

## if generate - Syntax

- "Conditional blocks"
- Comparable to if-else directives (e.g., `#if`, `#ifdef`) of the C preprocessor


212

- "Conditional blocks"
- Comparable to if-else directives (e.g., `#if`, `#ifdef`) of the C preprocessor
- If-generate syntax

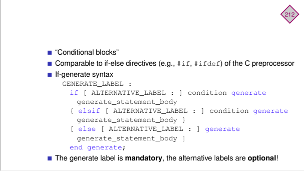
```
GENERATE_LABEL :
  if [ ALTERNATIVE_LABEL : ] condition generate
    generate_statement_body
  { elsif [ ALTERNATIVE_LABEL : ] condition generate
    generate_statement_body }
  [ else [ ALTERNATIVE_LABEL : ] generate
    generate_statement_body ]
  end generate;
```

An if-generate statement is introduced by a mandatory label followed by a colon and the keyword ”if”. Then the actual condition follows. This condition must be a static expression that returns a boolean value. As already alluded to, in this context static means that the compiler must be able to determine its value. It can, thus, for example not test the value of some signal. Oftentimes the condition will involve a generic parameter or some package constant. In any case, after the condition the ”generate” keyword introduces the actual generate statement body, which will be discussed in more detail on the next slide. As with regular if statements, after the actual ”if-branch” there can be an arbitrary number of ”else-if-branches”, followed by a single optional ”else-branch”. It is also fine if none of the conditions evaluate to true and none of the branches are actually active.

## if generate - Syntax

- "Conditional blocks"
- Comparable to if-else directives (e.g., `#if`, `#ifdef`) of the C preprocessor
- If-generate syntax

```
GENERATE_LABEL :
  if [ ALTERNATIVE_LABEL : ] condition generate
    generate_statement_body
  { elsif [ ALTERNATIVE_LABEL : ] condition generate
    generate_statement_body }
  [ else [ ALTERNATIVE_LABEL : ] generate
    generate_statement_body ]
  end generate;
```


212

- “Conditional blocks”
- Comparable to if-else directives (e.g., #if, #ifdef) of the C preprocessor
- If-generate syntax

```
GENERATE_LABEL :
  if [ ALTERNATIVE_LABEL : ] condition generate
    generate_statement_body
  { elsif [ ALTERNATIVE_LABEL : ] condition generate
    generate_statement_body }
  [ else [ ALTERNATIVE_LABEL : ] generate
    generate_statement_body ]
  end generate;
```

- The generate label is **mandatory**, the alternative labels are **optional**!

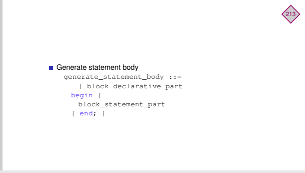
Note that every alternative-branch of the generate statement can be equipped with an optional label.

## if generate - Syntax

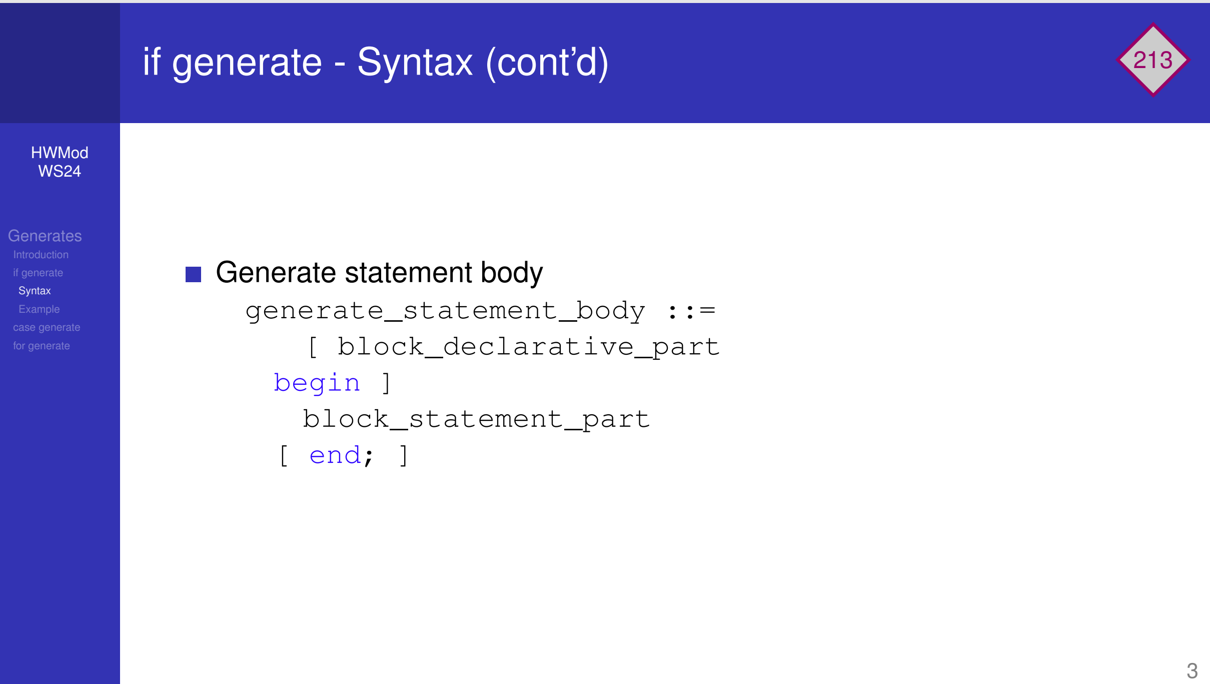
- “Conditional blocks”
- Comparable to if-else directives (e.g., #if, #ifdef) of the C preprocessor
- If-generate syntax

```
GENERATE_LABEL :
  if [ ALTERNATIVE_LABEL : ] condition generate
    generate_statement_body
  { elsif [ ALTERNATIVE_LABEL : ] condition generate
    generate_statement_body }
  [ else [ ALTERNATIVE_LABEL : ] generate
    generate_statement_body ]
  end generate;
```

- The generate label is **mandatory**, the alternative labels are **optional**!



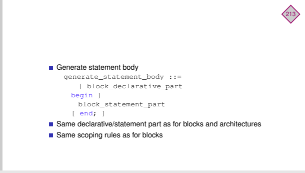
The generate-statement-body looks very similar to a block statement. It contains a declarative and a statement part separated by the ”begin” keyword and can optionally be explicitly terminated by an ”end” keyword. Notice that, if no declarations are needed, that is, if the body only consists of concurrent statements, the ”begin” keyword may be omitted.



## if generate - Syntax (cont'd)

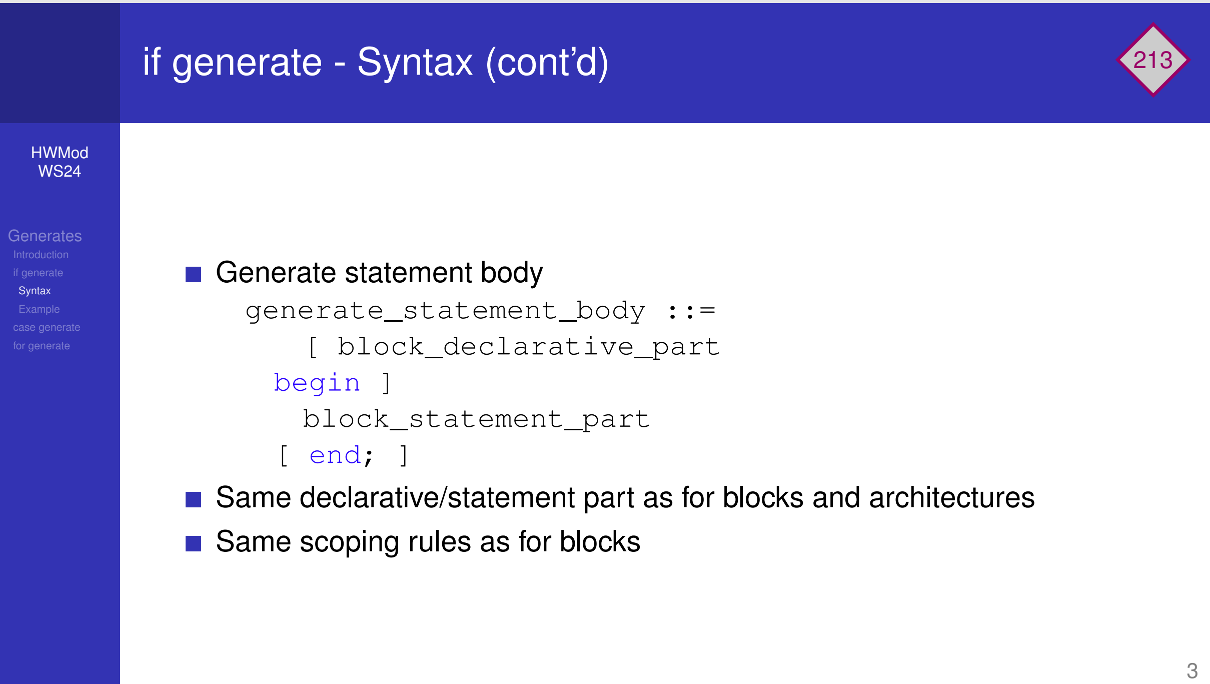
- Generate statement body

```
generate_statement_body ::=
    [ block_declarative_part
  begin ]
    block_statement_part
  [ end; ]
```


213

- Generate statement body
```
generate_statement_body ::=
    [ block_declarative_part
  begin ]
    block_statement_part
  [ end; ]
```
- Same declarative/statement part as for blocks and architectures
- Same scoping rules as for blocks

The declarative and statement parts can contain the same objects as blocks or architectures. Moreover, the same scoping rules apply as for blocks.



## if generate - Syntax (cont'd)

- Generate statement body
```
generate_statement_body ::=
    [ block_declarative_part
  begin ]
    block_statement_part
  [ end; ]
```
- Same declarative/statement part as for blocks and architectures
- Same scoping rules as for blocks

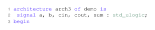
```
architecture arch3 of demo is
```

After this theoretical introduction to if-generates, let us look at some code to see them in action. In this code example we have a module that internally uses a full adder. Recall that we have already used a similar example in the lecture about block statements. However, this time we want to select between an exact version of the full adder circuit and an approximate one, which produces slightly inaccurate results but requires less hardware. This technique is widely used in approximate computing and a variety of different approximate adder circuits can be found in literature. In our example, the selection between the two versions shall be implemented with an if-generate statement, based on a string-type constant named `ADDER_TYPE`.

## if generate - Example

```
architecture arch3 of demo is
```

└─Generate Statements
  └─if generate
    └─**if generate - Example**

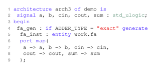


```
architecture arch3 of demo is
 signal a, b, cin, cout, sum : std_ulogic;
begin
```

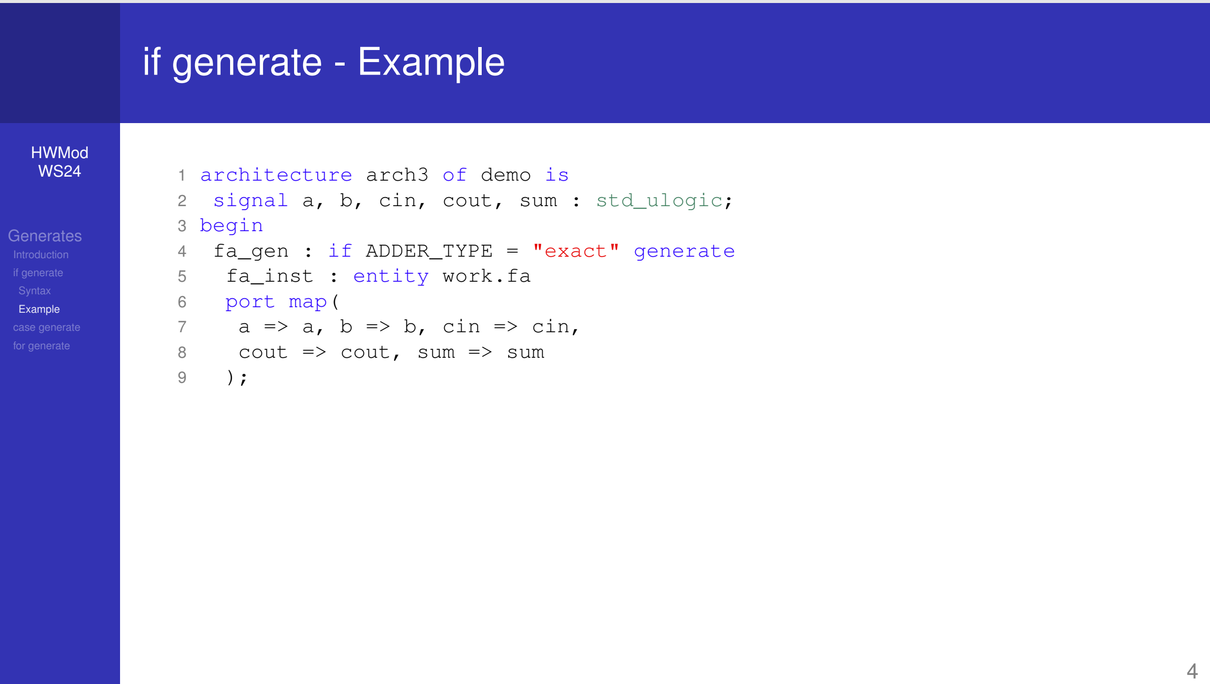
We start by declaring the usual full adder I/O signals.

## if generate - Example

```
architecture arch3 of demo is
 signal a, b, cin, cout, sum : std_ulogic;
begin
```

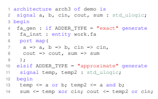


Next, we introduce the if-generate statement we need for selecting one of the two full adder versions. The if-branch of the generate statement shall be activated when the `ADDER_TYPE` constant has the value ”exact”. In this case we simply create an instance, of the full adder entity we are already familiar with.



## if generate - Example

```vhdl
architecture arch3 of demo is
 signal a, b, cin, cout, sum : std_ulogic;
begin
 fa_gen : if ADDER_TYPE = "exact" generate
  fa_inst : entity work.fa
  port map(
   a => a, b => b, cin => cin,
   cout => cout, sum => sum
  );
```

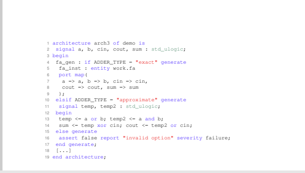


```vhdl
architecture arch3 of demo is
 signal a, b, cin, cout, sum : std_ulogic;
begin
 fa_gen : if ADDER_TYPE = "exact" generate
  fa_inst : entity work.fa
  port map(
   a => a, b => b, cin => cin,
   cout => cout, sum => sum
  );
 elsif ADDER_TYPE = "approximate" generate
  signal temp, temp2 : std_ulogic;
 begin
  temp <= a or b; temp2 <= a and b;
  sum <= temp xor cin; cout <= temp2 or cin;
```

In the else-if-branch we use concurrent statements to implement some version of an approximate adder.

## if generate - Example

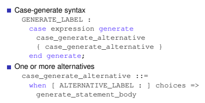
```vhdl
architecture arch3 of demo is
 signal a, b, cin, cout, sum : std_ulogic;
begin
 fa_gen : if ADDER_TYPE = "exact" generate
  fa_inst : entity work.fa
  port map(
   a => a, b => b, cin => cin,
   cout => cout, sum => sum
  );
 elsif ADDER_TYPE = "approximate" generate
  signal temp, temp2 : std_ulogic;
 begin
  temp <= a or b; temp2 <= a and b;
  sum <= temp xor cin; cout <= temp2 or cin;
```



Should the final else-branch be entered, we use an assertion with the severity level failure to raise a compilation error, because the adder-type constant did not match any of the two expected values. You might want to pause the video here, to really understand the presented code.

## if generate - Example

```vhdl
architecture arch3 of demo is
 signal a, b, cin, cout, sum : std_ulogic;
begin
 fa_gen : if ADDER_TYPE = "exact" generate
  fa_inst : entity work.fa
  port map(
   a => a, b => b, cin => cin,
   cout => cout, sum => sum
  );
 elsif ADDER_TYPE = "approximate" generate
  signal temp, temp2 : std_ulogic;
 begin
  temp <= a or b; temp2 <= a and b;
  sum <= temp xor cin; cout <= temp2 or cin;
 else generate
  assert false report "invalid option" severity failure;
 end generate;
 [...]
end architecture;
```


Case-generate syntax
```
GENERATE_LABEL :
  case expression generate
    case_generate_alternative
    { case_generate_alternative }
  end generate;
```
One or more alternatives
```
case_generate_alternative ::=
  when [ ALTERNATIVE_LABEL : ] choices =>
    generate_statement_body
```

The case-generate statement is very similar to the if-generate-statement. It evaluates a single expression and then activates exactly one of the when-clauses.

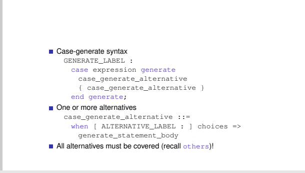
## case generate

- Case-generate syntax

```
GENERATE_LABEL :
  case expression generate
    case_generate_alternative
    { case_generate_alternative }
  end generate;
```

- One or more alternatives

```
case_generate_alternative ::=
  when [ ALTERNATIVE_LABEL : ] choices =>
    generate_statement_body
```


■ Case-generate syntax
```
GENERATE_LABEL :
  case expression generate
    case_generate_alternative
    { case_generate_alternative }
  end generate;
```
■ One or more alternatives
```
case_generate_alternative ::=
  when [ ALTERNATIVE_LABEL : ] choices =>
    generate_statement_body
```
■ All alternatives must be covered (recall `others`)!

Just as regular case statements used in processes, case-generates must also cover all possible alternatives to which the expression can evaluate. For that purpose the last when-clause may use the ”others” keyword. Note that we just want to quickly show you its syntax definition here, without going into too much detail, as by now it should be quite clear how it works. Feel free to pause the video to study the syntax.

## case generate

- Case-generate syntax

```
GENERATE_LABEL :
  case expression generate
    case_generate_alternative
    { case_generate_alternative }
  end generate;
```

- One or more alternatives

```
case_generate_alternative ::=
  when [ ALTERNATIVE_LABEL : ] choices =>
    generate_statement_body
```

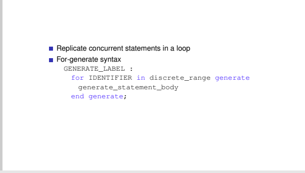
- All alternatives must be covered (recall `others`)!

■ Replicate concurrent statements in a loop

Finally, let us discuss the for-generate statement. As already mentioned, in VHDL the for-generate statement is used to create multiple instances of a design component or to replicate hardware structures systematically within a design. It allows for repetitive generation of components or logic constructs by having a variable loop over a defined range, which is especially useful when you need a large number of similar components, such as in cases where you are creating multiple bits of a register or implementing an array of identical processing elements.

## for generate - Syntax

- Replicate concurrent statements in a loop

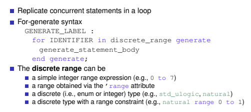


The for-generate syntax is quite straight-forward. As with the other generate variants, it starts with a mandatory label which is then followed by a colon and the keyword ”for”. Then an identifier has to be specified which represents the loop variable and is accessible within the body of the generate statement.

## for generate - Syntax

- Replicate concurrent statements in a loop
- For-generate syntax

```
GENERATE_LABEL :
  for IDENTIFIER in discrete_range generate
    generate_statement_body
  end generate;
```


- Replicate concurrent statements in a loop
- For-generate syntax

```
GENERATE_LABEL :
  for IDENTIFIER in discrete_range generate
    generate_statement_body
  end generate;
```

- The **discrete range** can be
  - a simple integer range expression (e.g., `0 to 7`)
  - a range obtained via the `'range` attribute
  - a discrete (i.e., enum or integer) type (e.g., `std_ulogic`, `natural`)
  - a discrete type with a range constraint (e.g., `natural range 0 to 1`)

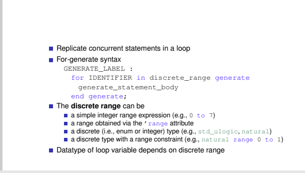
After the keyword ”in”, a discrete range must be specified for the for-generate statement to loop over. There are quite a lot of ways to specify a range here. However, most often you will probably use the simple integer range or the range attribute.

## for generate - Syntax

- Replicate concurrent statements in a loop
- For-generate syntax

```
GENERATE_LABEL :
  for IDENTIFIER in discrete_range generate
    generate_statement_body
  end generate;
```

- The **discrete range** can be
  - a simple integer range expression (e.g., `0 to 7`)
  - a range obtained via the `'range` attribute
  - a discrete (i.e., enum or integer) type (e.g., `std_ulogic`, `natural`)
  - a discrete type with a range constraint (e.g., `natural range 0 to 1`)



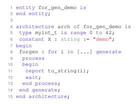
Please note that the datatype of the loop variable depends on the discrete range. However, most of the time it will be an integer.

## for generate - Syntax

- Replicate concurrent statements in a loop
- For-generate syntax

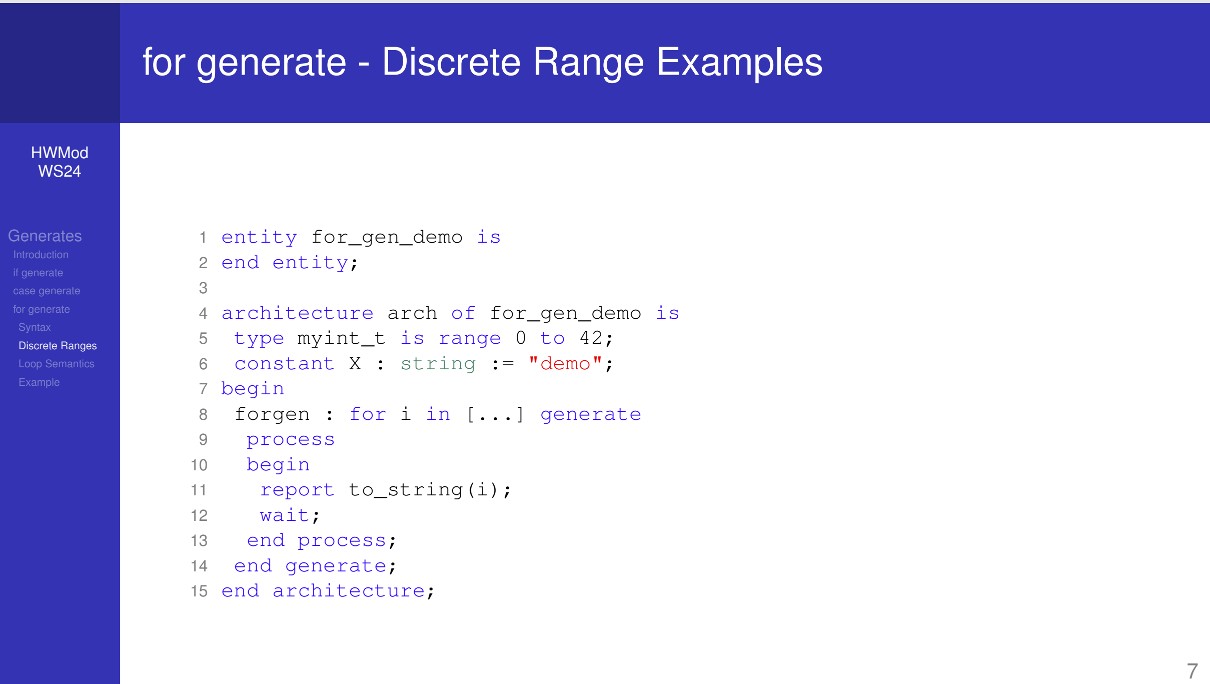
```
GENERATE_LABEL :
  for IDENTIFIER in discrete_range generate
   generate_statement_body
  end generate;
```

- The **discrete range** can be
  - a simple integer range expression (e.g., `0 to 7`)
  - a range obtained via the `'range` attribute
  - a discrete (i.e., enum or integer) type (e.g., `std_ulogic`, `natural`)
  - a discrete type with a range constraint (e.g., `natural range 0 to 1`)
- Datatype of loop variable depends on discrete range



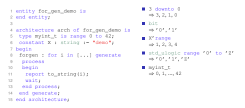
```vhdl
entity for_gen_demo is
end entity;

architecture arch of for_gen_demo is
 type myint_t is range 0 to 42;
 constant X : string := "demo";
begin
 forgen : for i in [...] generate
  process
  begin
   report to_string(i);
   wait;
  end process;
 end generate;
end architecture;
```

This slide shows some examples of how the discrete range can look in practice. For that purpose, consider the `for_gen_demo` module containing a for-generate statement that utilizes a single process in its statement part to print the value of the loop variable. Notice that we have replaced the discrete range with a placeholder represented by three dots.

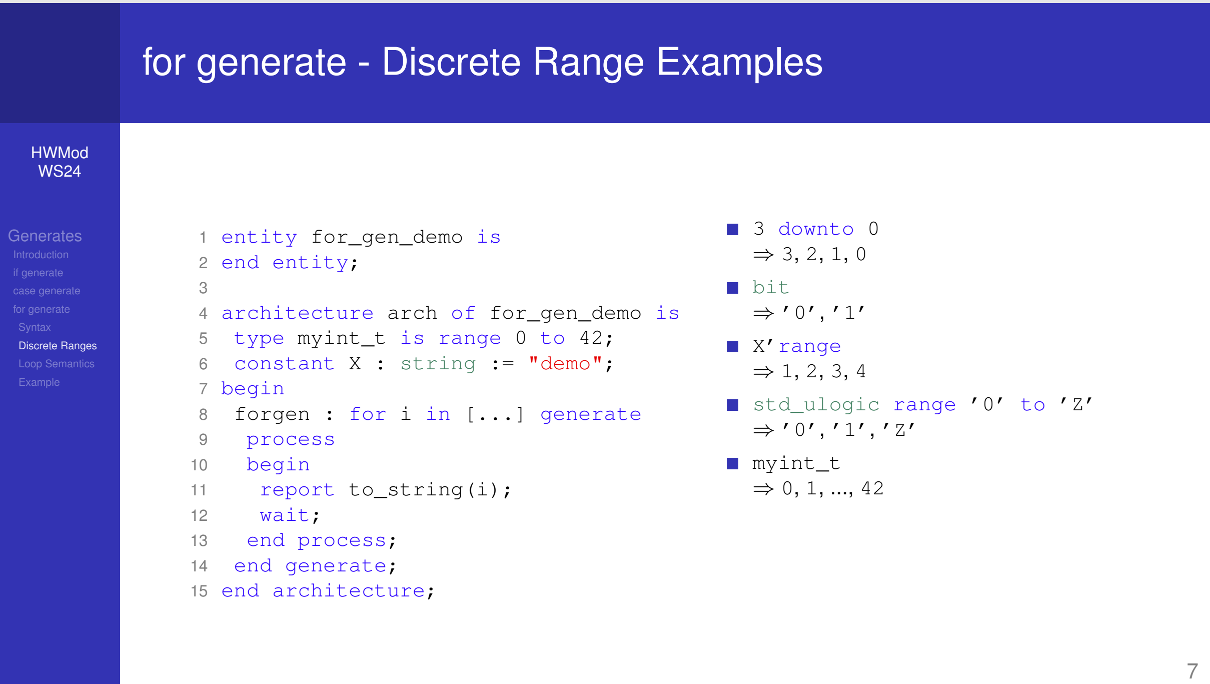


## for generate - Discrete Range Examples

```vhdl
entity for_gen_demo is
end entity;

architecture arch of for_gen_demo is
 type myint_t is range 0 to 42;
 constant X : string := "demo";
begin
 forgen : for i in [...] generate
  process
  begin
   report to_string(i);
   wait;
  end process;
 end generate;
end architecture;
```



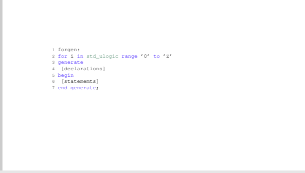
The right side of the slide now shows the code's output when running it in a simulator using different discrete ranges. The comma-separated lists after the arrows show the string output lines as produced by the report statements in the different processes of the generate body. However, note that, depending on the execution order of the processes in the simulator, the sequence of the output values might be different. You may want to pause the video at this point, to really understand all the shown outputs.



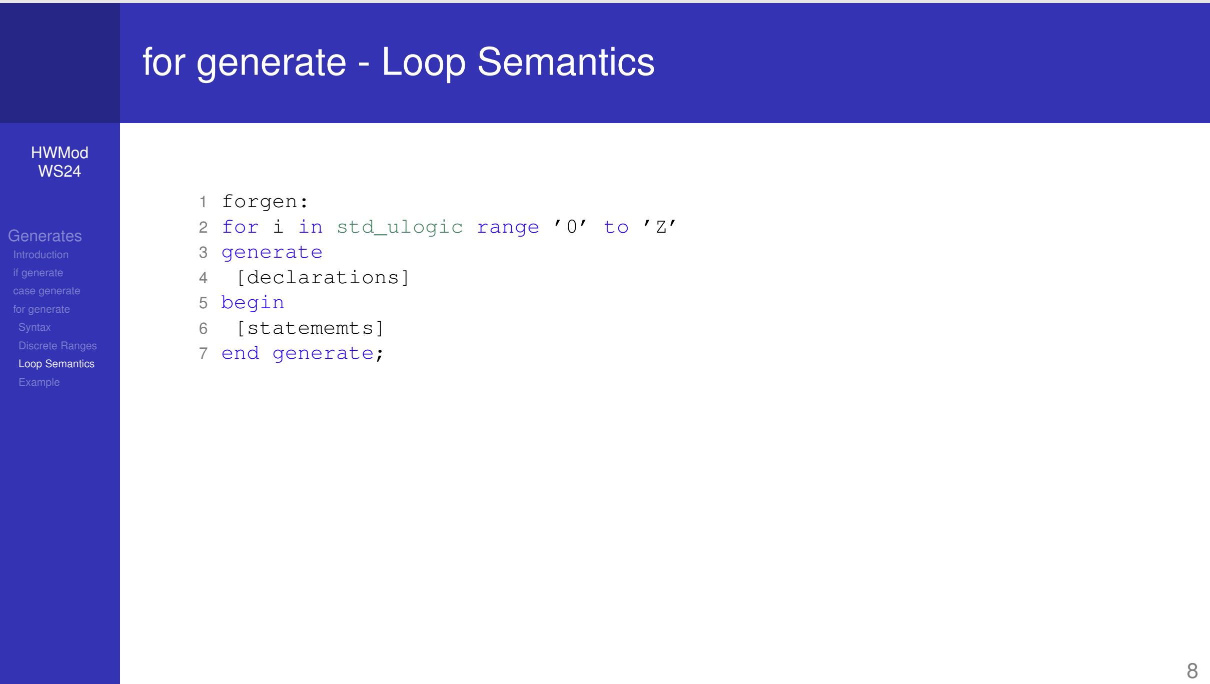
## for generate - Discrete Range Examples

```
entity for_gen_demo is
end entity;

architecture arch of for_gen_demo is
 type myint_t is range 0 to 42;
 constant X : string := "demo";
begin
 forgen : for i in [...] generate
  process
  begin
   report to_string(i);
   wait;
  end process;
 end generate;
end architecture;
```

- `3 downto 0`
  ⇒ 3, 2, 1, 0
- `bit`
  ⇒ '0', '1'
- `X'range`
  ⇒ 1, 2, 3, 4
- `std_ulogic range '0' to 'Z'`
  ⇒ '0', '1', 'Z'
- `myint_t`
  ⇒ 0, 1, ..., 42



In order to illustrate the semantics of the for-generate statement, let us pick one of the discrete ranges of the previous slide and see how the loop is unrolled and interpreted. For that purpose we will convert the for-generate into an equivalent piece of code consisting of block statements. In our example we will consider the range 0 to Z of the `std_ulogic` enumeration type. Recall that this range comprises the three values 0, 1 and Z.



## for generate - Loop Semantics

```
forgen:
for i in std_ulogic range '0' to 'Z'
generate
 [declarations]
begin
 [statememts]
end generate;
```

```
forgen:
for i in std_ulogic range '0' to 'Z'
generate
 [declarations]
begin
 [statememts]
end generate;
```

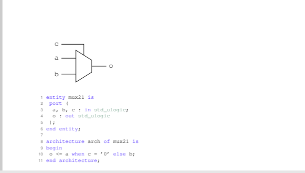
```
\forgen('0')\ : block
 constant i : std_ulogic := '0';
 [declarations]
begin
 [statements]
end block;

\forgen('1')\ : block
 constant i : std_ulogic := '1';
 [declarations]
begin
 [statements]
end block;

\forgen('Z')\ : block
 constant i : std_ulogic := 'Z';
 [declarations]
begin
 [statements]
end block;
```

Since the specified range consists of three values, we need three blocks to represent it. You can think of the loop variable "i" as an additional constant declared in the beginning of each of those blocks. The declaration and statement part of for-generate body is simply copied to each of the blocks. The block names are chosen according to how QuestaSim would identify them in simulation.

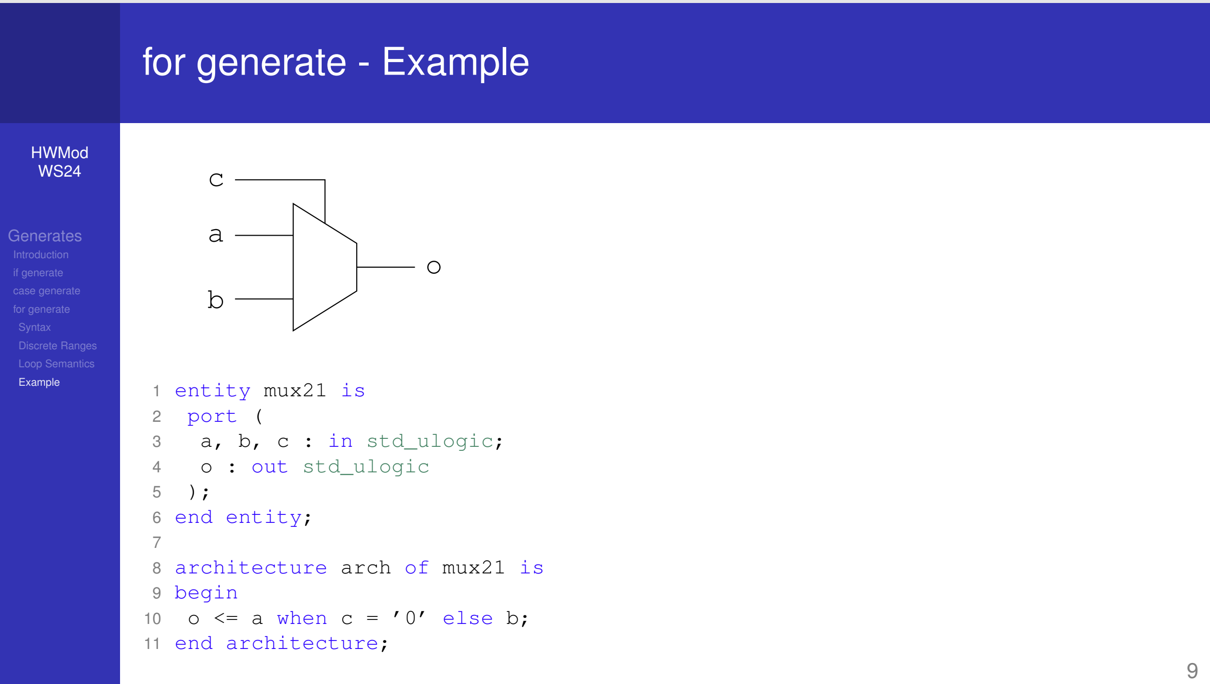
## for generate - Loop Semantics

```
forgen:
for i in std_ulogic range '0' to 'Z'
generate
 [declarations]
begin
 [statememts]
end generate;
```

```
\forgen('0')\ : block
 constant i : std_ulogic := '0';
 [declarations]
begin
 [statements]
end block;

\forgen('1')\ : block
 constant i : std_ulogic := '1';
 [declarations]
begin
 [statements]
end block;

\forgen('Z')\ : block
 constant i : std_ulogic := 'Z';
 [declarations]
begin
 [statements]
end block;
```

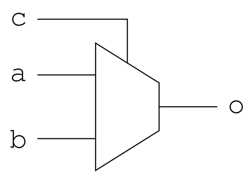


Finally, let’s look at a real-world example where a for-generate statement comes in handy. Let’s say we have a simple 2-to-1 multiplexer, such as the one shown on the left side of the slide. Both of its data inputs ”a” and ”b”, as well as its data output ”o” are single bit signals of type `std_ulogic`. We now want to use this module to create a multiplexer that can multiplex between input signals of width ”N”.

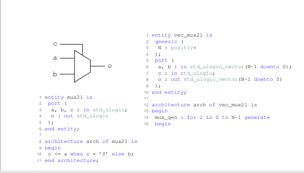


## for generate - Example

HWMod
WS24

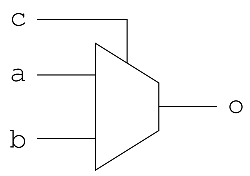


```
entity mux21 is
 port (
  a, b, c : in std_ulogic;
  o : out std_ulogic
 );
end entity;

architecture arch of mux21 is
begin
 o <= a when c = '0' else b;
end architecture;
```



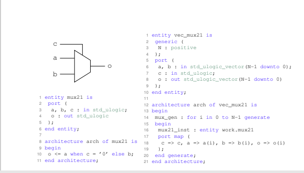
The resulting module shall be called `vec_mux21`. Its entity declaration should not look too surprising, as we have already encountered this module in a previous lecture. It has the same input and output signals as the single-bit multiplexer on the left, with the difference that the data signals "a", "b" and "o" are now "N"-bit wide vectors.

## for generate - Example

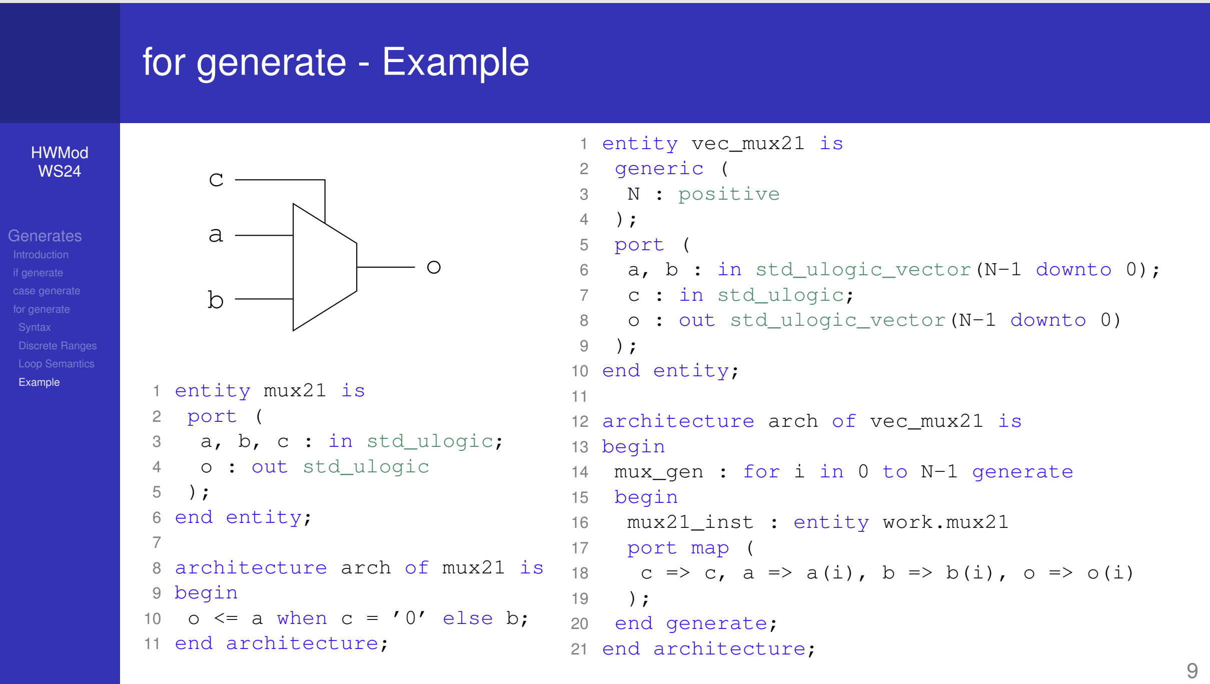


```
entity mux21 is
 port (
  a, b, c : in std_ulogic;
  o : out std_ulogic
 );
end entity;

architecture arch of mux21 is
begin
 o <= a when c = '0' else b;
end architecture;
```

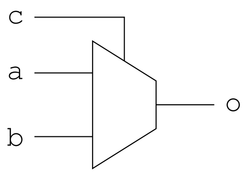
```
entity vec_mux21 is
 generic (
  N : positive
 );
 port (
  a, b : in std_ulogic_vector(N-1 downto 0);
  c : in std_ulogic;
  o : out std_ulogic_vector(N-1 downto 0)
 );
end entity;

architecture arch of vec_mux21 is
begin
 mux_gen : for i in 0 to N-1 generate
 begin
```



In its architecture we are now going to create ”N” instances of the `mux21` entity – one for each of the ”N” bits in the input and output signals. For that purpose we use a range from 0 to ”N” minus 1.

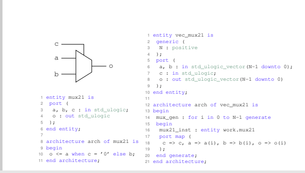


## for generate - Example



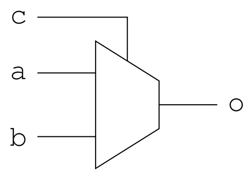
```
entity mux21 is
 port (
  a, b, c : in std_ulogic;
  o : out std_ulogic
 );
end entity;

architecture arch of mux21 is
begin
 o <= a when c = '0' else b;
end architecture;
```

```
entity vec_mux21 is
 generic (
  N : positive
 );
 port (
  a, b : in std_ulogic_vector(N-1 downto 0);
  c : in std_ulogic;
  o : out std_ulogic_vector(N-1 downto 0)
 );
end entity;

architecture arch of vec_mux21 is
begin
 mux_gen : for i in 0 to N-1 generate
 begin
  mux21_inst : entity work.mux21
  port map (
   c => c, a => a(i), b => b(i), o => o(i)
  );
 end generate;
end architecture;
```



During instantiation we can then use the loop-variable ”i” to access the individual bits of the vector signals ”a”, ”b” and ”o”. Hence, in total the for-generate statement create ”N” instances of the single-bit multiplexer.

## for generate - Example



```
entity mux21 is
 port (
  a, b, c : in std_ulogic;
  o : out std_ulogic
 );
end entity;

architecture arch of mux21 is
begin
 o <= a when c = '0' else b;
end architecture;
```

```
entity vec_mux21 is
 generic (
  N : positive
 );
 port (
  a, b : in std_ulogic_vector(N-1 downto 0);
  c : in std_ulogic;
  o : out std_ulogic_vector(N-1 downto 0)
 );
end entity;

architecture arch of vec_mux21 is
begin
 mux_gen : for i in 0 to N-1 generate
 begin
  mux21_inst : entity work.mux21
  port map (
   c => c, a => a(i), b => b(i), o => o(i)
  );
 end generate;
end architecture;
```

Lecture Complete!

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.

HWMod
WS24

# Lecture Complete!