Hardware Modeling [VU] (191.011)
– WS24 –
IEEE 1164 Package

Florian Huemer & Sebastian Wiedemann & Dylan Baumann

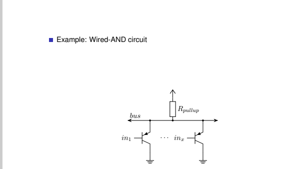
WS 2024/25

In this lecture, we will introduce the `std_logic_1164` package, that implements the IEEE 1164 standard. It contains the types that are virtually always used when describing hardware in VHDL.

Hardware Modeling [VU] (191.011)
– WS24 –
IEEE 1164 Package

Florian Huemer & Sebastian Wiedemann & Dylan Baumann

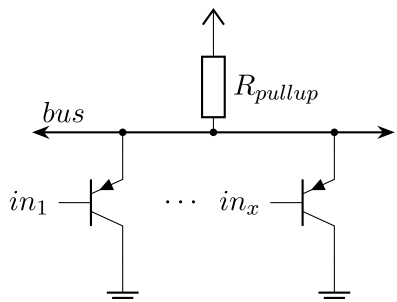
WS 2024/25

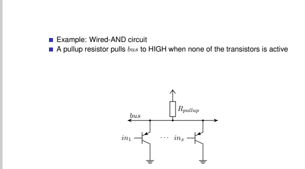


Before we start our discussion about the additional types the package introduces, let us consider two examples that stress the need for them. The first one is a bus with wired AND topology. This can, for instance, be found in the I^{2}C protocol. The figure on the slide shows a schematic of such a circuit.

Motivation | Wired-AND

- Example: Wired-AND circuit

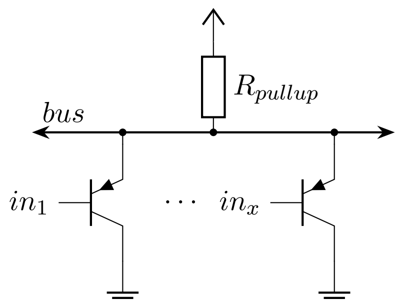


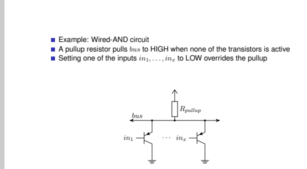


The pullup resistor ensures that the bus line exhibits a valid logical value at all times, by connecting it to the voltage corresponding to a logical high.

Motivation | Wired-AND

- Example: Wired-AND circuit
- A pullup resistor pulls bus to HIGH when none of the transistors is active

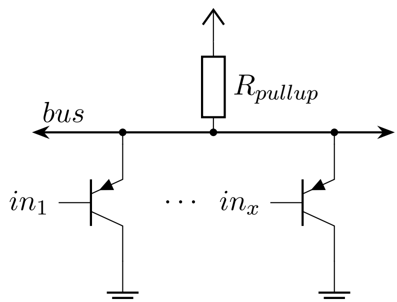


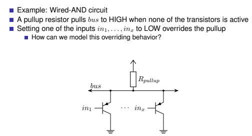


When a bus participant wants to transmit a logical low, it sets the input voltage of its respective bus driver to low, which will effectively pull the bus voltage to ground, overriding the pullup resistor. Since the bus exhibits a logical low whenever any of the bus driver inputs is set to low, this type of connection leads to an AND behavior.

Motivation | Wired-AND

- Example: Wired-AND circuit
- A pullup resistor pulls bus to HIGH when none of the transistors is active
- Setting one of the inputs $in_1, \ldots, in_x$ to LOW overrides the pullup

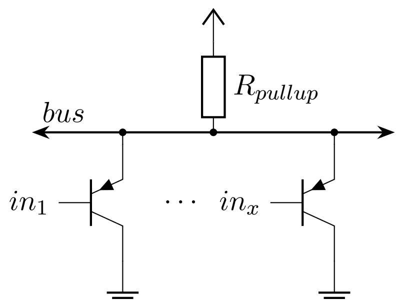


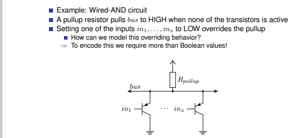


Now assume you are to model this circuit using VHDL. How would you *do* it?

Motivation | Wired-AND

- Example: Wired-AND circuit
- A pullup resistor pulls bus to HIGH when none of the transistors is active
- Setting one of the inputs $in_1, \ldots, in_x$ to LOW overrides the pullup
  - How can we model this overriding behavior?

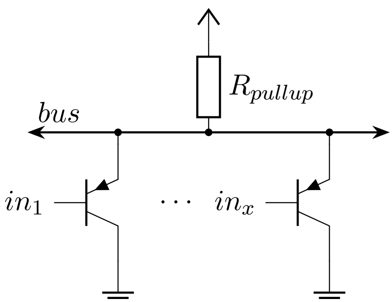


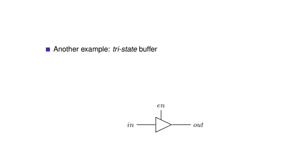


We clearly cannot use the basic Boolean type and encode the desired circuit behavior - we would just end up describing an AND gate since Boolean logic has no concept of driving strength. Therefore, what we need is a means to model this property of an acitvely driven signal. In particular, we need the *weak* logical high provided by the pullup resistor, and the *strong* logical low provided by the bus drivers.

Motivation | Wired-AND

- Example: Wired-AND circuit
- A pullup resistor pulls bus to HIGH when none of the transistors is active
- Setting one of the inputs $in_1, \ldots, in_x$ to LOW overrides the pullup
  - How can we model this overriding behavior?
  - ⇒ To encode this we require more than Boolean values!

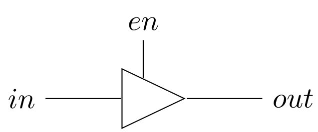


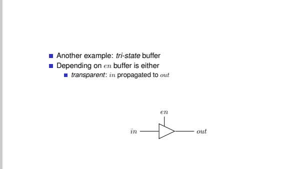


Our second example for the shortcomings of Boolean logic for describing hardware is a *tri-state* buffer. This special kind of buffer circuit, shown on the slide, has two states, controlled via the enable signal, referred to as `en`.

Motivation | Tri-State Buffer

- Another example: *tri-state* buffer

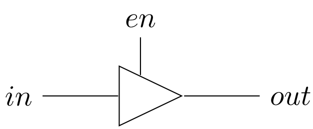


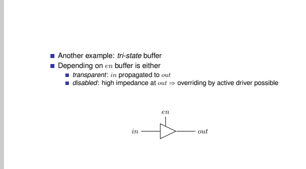


If the enable signal is high, the buffer will be transparent. This means that it will simply propagate the logical values applied at its input to its output.

Motivation | Tri-State Buffer

- Another example: *tri-state* buffer
- Depending on en buffer is either
  - *transparent*: in propagated to out

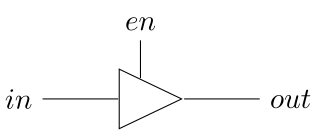




However, if the buffer is disabled, it will be in a so-called high-impedance state. In this state the buffer input is not connected to the output, allowing the output to be overridden by active drivers.

Motivation | Tri-State Buffer

- Another example: *tri-state* buffer
- Depending on en buffer is either
  - *transparent*: in propagated to out
  - *disabled*: high impedance at out $\Rightarrow$ overriding by active driver possible



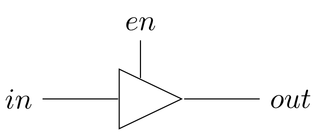
Another example: *tri-state* buffer
Depending on en buffer is either
transparent: in propagated to out
disabled: high impedance at out ⇒ overriding by active driver possible
⇒ We cannot model this with Boolean values alone!

Similar to the wired AND-topology, this can be used to build a bus that is shared by multiple participants. However, as before, we face the problem that Boolean logic is not expressive enough to model this additional high-impedance state.

Motivation | Tri-State Buffer

- Another example: *tri-state* buffer
- Depending on en buffer is either
  - *transparent*: in propagated to out
  - *disabled*: high impedance at out ⇒ overriding by active driver possible

⇒ We cannot model this with Boolean values alone!



Recall from *Digital Design* lecture: 9-valued logic
Contains values for different driver strength / impedance
Also values useful for simulation and synthesis

At this point, you might recall the nine-valued logic from the Digital Design lecture. Instead of the typical Boolean low and high values, this special logic comes with values for different driver strength and impedance, as well as values useful for simulation and synthesis.

The IEEE std_logic_1164 package

- Recall from *Digital Design* lecture: 9-valued logic
  - Contains values for different driver strength / impedance
  - Also values useful for simulation and synthesis

■ Recall from *Digital Design* lecture: 9-valued logic
 ■ Contains values for different driver strength / impedance
 ■ Also values useful for simulation and synthesis
 ■ IEEE 1164 standard for VHDL

The IEEE standardized this special value system for VHDL in the 1164 standard in 1993.

The IEEE std_logic_1164 package

- Recall from *Digital Design* lecture: 9-valued logic
  - Contains values for different driver strength / impedance
  - Also values useful for simulation and synthesis
  - IEEE 1164 standard for VHDL

327

- Recall from *Digital Design* lecture: 9-valued logic
  - Contains values for different driver strength / impedance
  - Also values useful for simulation and synthesis
  - IEEE 1164 standard for VHDL
- Implemented in the `std_logic_1164` package

In addition to defining it, the IEEE also provides an open-source implementation of this standard in the form of the `std_logic_1164` package. You can have a look at this implementation by clicking the icon on the slide.

The IEEE std_logic_1164 package

- Recall from *Digital Design* lecture: 9-valued logic
  - Contains values for different driver strength / impedance
  - Also values useful for simulation and synthesis
  - IEEE 1164 standard for VHDL
- Implemented in the `std_logic_1164` package

Recall from *Digital Design* lecture: 9-valued logic
Contains values for different driver strength / impedance
Also values useful for simulation and synthesis
IEEE 1164 standard for VHDL
Implemented in the std_logic_1164 package
Must be imported via
library ieee;
use ieee.std_logic_1164.all;

Note that, similar to including some standard C header file, or a Java module, the respective library and the package must be imported before you have access to the additional functionality. As shown on the slide, this first requires to import the IEEE library and then the particular required package of this library.

The IEEE std_logic_1164 package

HWMod
WS24

IEEE 1164
Motivation
Standard
Package
Value System
VHDL Types
Resolution
Operators
Vector Types
Conclusion

- Recall from *Digital Design* lecture: 9-valued logic
  - Contains values for different driver strength / impedance
  - Also values useful for simulation and synthesis
  - IEEE 1164 standard for VHDL
- Implemented in the `std_logic_1164` package IEEE SA OPEN
- Must be imported via

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

Recall from *Digital Design* lecture: 9-valued logic
Contains values for different driver strength / impedance
Also values useful for simulation and synthesis
IEEE 1164 standard for VHDL
Implemented in the std_logic_1164 package
Must be imported via
library ieee;
use ieee.std_logic_1164.all;
Essentially introduces two new, 9-valued, types:
Unresolved std_ulogic
Resolved std_logic

After such an import, you essentially gain access to two new types: `std_ulogic` and `std_logic`, as well as to operations defined for them. While the two types are named similarly, will see shortly that they actually behave quite different from another, We coin these different behaviors as *unresolved* and *resolved*.

The IEEE std_logic_1164 package

- Recall from *Digital Design* lecture: 9-valued logic
  - Contains values for different driver strength / impedance
  - Also values useful for simulation and synthesis
  - IEEE 1164 standard for VHDL
- Implemented in the `std_logic_1164` package IEEE SA OPEN
- Must be imported via

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

- Essentially introduces two new, 9-valued, types:
  - Unresolved `std_ulogic` IEEE SA OPEN
  - Resolved `std_logic` IEEE SA OPEN

The standard defines nine values and example use cases

Value	Name	Example Use Case

Before we continue with the specifics of the two types, let us briefly introduce the nine values which `std_ulogic` and `std_logic` share, as well as the example uses cases the standard mentions for them.

IEEE 1164 Value System

The standard defines nine values and example use cases

Value	Name	Example Use Case

The standard defines nine values and example use cases

Value	Name	Example Use Case
'U'	Uninitialized State	Used as default value

The uninitialized state, 'U', can be used to detect signals that have not been changed since the simulation start. Per default, all instances of the `std_ulogic` and `std_logic` types are initialized to this value.

IEEE 1164 Value System

The standard defines nine values and example use cases

Value	Name	Example Use Case
'U'	Uninitialized State	Used as default value

The standard defines nine values and example use cases

Value	Name	Example Use Case
'U'	Uninitialized State	Used as default value
'X'	Strong Unknown	Bus contention, error condition

562

The value `'X'` is used to express conflicting drivers or errors that prevent the simulator from determining one of the other eight values. An example is the case of multiple drivers of the same strength applying different values to the same signal. Since the drivers are of equal strength, none will dominate the other one and the resulting value is thus unknown.

IEEE 1164 Value System

The standard defines nine values and example use cases

Value	Name	Example Use Case
'U'	Uninitialized State	Used as default value
'X'	Strong Unknown	Bus contention, error condition

The standard defines nine values and example use cases

Value	Name	Example Use Case
'U'	Uninitialized State	Used as default value
'X'	Strong Unknown	Bus contention, error condition
'0'	Strong LOW	Active driver to LOW
'1'	Strong HIGH	Active driver to HIGH

The values '0' and '1' reflect the classic Boolean logic values.

IEEE 1164 Value System

The standard defines nine values and example use cases

Value	Name	Example Use Case
'U'	Uninitialized State	Used as default value
'X'	Strong Unknown	Bus contention, error condition
'0'	Strong LOW	Active driver to LOW
'1'	Strong HIGH	Active driver to HIGH

The value ’Z’ is associated to a high impedance. It is of use in cases like the initial tri-state buffer example.

IEEE 1164 Value System

The standard defines nine values and example use cases

Value	Name	Example Use Case
’U’	Uninitialized State	Used as default value
’X’	Strong Unknown	Bus contention, error condition
’0’	Strong LOW	Active driver to LOW
’1’	Strong HIGH	Active driver to HIGH
’Z’	High Impedance	Tri-state buffer output

562

The standard defines nine values and example use cases

Value	Name	Example Use Case
'U'	Uninitialized State	Used as default value
'X'	Strong Unknown	Bus contention, error condition
'0'	Strong LOW	Active driver to LOW
'1'	Strong HIGH	Active driver to HIGH
'Z'	High Impedance	Tri-state buffer output
'W'	Weak Unknown	Bus terminator
'L'	Weak LOW	Pull down resistor
'H'	Weak HIGH	Pull up resistor

Furthermore, there are the weak unknown, low, and high values, referred to as `'W'`, `'L'` respectively `'H'`. These values are used to model different driver strengths, therefore allowing to express overriding behavior as required in the initial wired AND example.

IEEE 1164 Value System

The standard defines nine values and example use cases

Value	Name	Example Use Case
'U'	Uninitialized State	Used as default value
'X'	Strong Unknown	Bus contention, error condition
'0'	Strong LOW	Active driver to LOW
'1'	Strong HIGH	Active driver to HIGH
'Z'	High Impedance	Tri-state buffer output
'W'	Weak Unknown	Bus terminator
'L'	Weak LOW	Pull down resistor
'H'	Weak HIGH	Pull up resistor

The standard defines nine values and example use cases

Value	Name	Example Use Case
'U'	Uninitialized State	Used as default value
'X'	Strong Unknown	Bus contention, error condition
'0'	Strong LOW	Active driver to LOW
'1'	Strong HIGH	Active driver to HIGH
'Z'	High Impedance	Tri-state buffer output
'W'	Weak Unknown	Bus terminator
'L'	Weak LOW	Pull down resistor
'H'	Weak HIGH	Pull up resistor
'-'	Don't care	Useful for synthesis and modeling

Finally, the standard also defines a don't care value, referred to via a `'-'` symbol. This is useful in cases where not all inputs, or states, of the modelled circuit can actually occur, allowing the synthesis tool to perform some optimizations. Let us now look at the types provided by the `std_logic_1164` package.

IEEE 1164 Value System

The standard defines nine values and example use cases

Value	Name	Example Use Case
'U'	Uninitialized State	Used as default value
'X'	Strong Unknown	Bus contention, error condition
'0'	Strong LOW	Active driver to LOW
'1'	Strong HIGH	Active driver to HIGH
'Z'	High Impedance	Tri-state buffer output
'W'	Weak Unknown	Bus terminator
'L'	Weak LOW	Pull down resistor
'H'	Weak HIGH	Pull up resistor
'-'	Don't care	Useful for synthesis and modeling

■ Simple enumeration type with nine values
type std_ulogic is ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-');

First we will discuss the `std_ulogic` type. In principle, this type is nothing more than an enumeration type consisting of the previously mentioned nine values. Observe how the value U is the enum's first value, leading to the mentioned default initialization.

IEEE 1164 std_ulogic Type

- Simple enumeration type with nine values

```
type std_ulogic is ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-');
```

IEEE SA OPEN

1

Simple enumeration type with nine values
type std_ulogic is ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-');
Unresolved: Only supports signals with **single** driver

As we stated before, the `std_ulogic` type is *unresolved*, hence the *u* in its name. This means, that signals of this type only permit a **single** driver.

IEEE 1164 std_ulogic Type

- Simple enumeration type with nine values

```
type std_ulogic is ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-');
```

IEEE SA OPEN

- *Unresolved*: Only supports signals with **single** driver

1

A violation of this, meaning multiple drivers for a single signals, will be detected by simulators and reported when elaborating VHDL code.

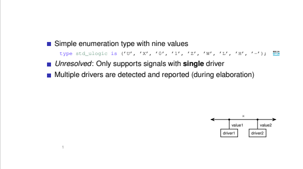
IEEE 1164 std_ulogic Type

- Simple enumeration type with nine values

```
type std_ulogic is ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-');
```

- *Unresolved*: Only supports signals with **single** driver
- Multiple drivers are detected and reported (during elaboration)

1



To illustrate this, consider the example on the slide where two drivers share a `std_ulogic` signal `x`.

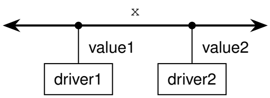
IEEE 1164 std_ulogic Type

- Simple enumeration type with nine values

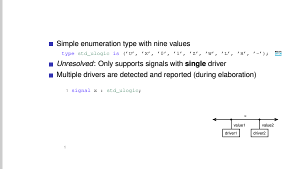
```
type std_ulogic is ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-');
```

IEEE SA OPEN

- *Unresolved*: Only supports signals with **single** driver
- Multiple drivers are detected and reported (during elaboration)



1



In VHDL this would correspond to a signal `x` of type `std_ulogic` being declared first, which will then be shared by the drivers.

IEEE 1164 std_ulogic Type

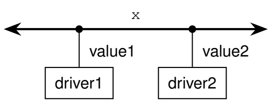
- Simple enumeration type with nine values

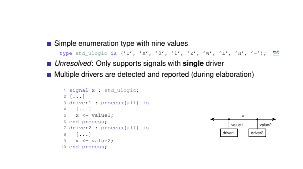
```
type std_ulogic is ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-');
```

IEEE SA OPEN

- *Unresolved*: Only supports signals with **single** driver
- Multiple drivers are detected and reported (during elaboration)

```
signal x : std_ulogic;
```





Said drivers could be implemented as distinct processes, as shown on the slide.

IEEE 1164 std_ulogic Type

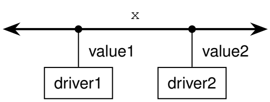
- Simple enumeration type with nine values

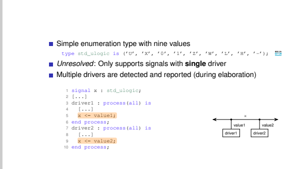
```
type std_ulogic is ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-');
```

IEEE SA OPEN

- *Unresolved*: Only supports signals with **single** driver
- Multiple drivers are detected and reported (during elaboration)

```
signal x : std_ulogic;
[...]
driver1 : process(all) is
  [...]
  x <= value1;
end process;
driver2 : process(all) is
  [...]
  x <= value2;
end process;
```





Note how both of the processes contain an assignment to `x`, thus resulting in conflicting drivers.

IEEE 1164 std_ulogic Type

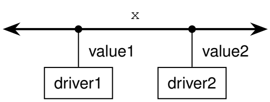
- Simple enumeration type with nine values

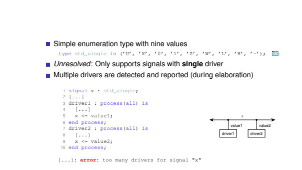
```
type std_ulogic is ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-');
```

IEEE SA OPEN

- *Unresolved*: Only supports signals with **single** driver
- Multiple drivers are detected and reported (during elaboration)

```
signal x : std_ulogic;
[...]
driver1 : process(all) is
  [...]
  x <= value1;
end process;
driver2 : process(all) is
  [...]
  x <= value2;
end process;
```





If you were now to try simulating this circuit, you would observe that the simulator produces an error like the one shown on the bottom of the slide. As mentioned, the reason is of course that a signal of type `std_ulogic` does not allow multiple drivers.

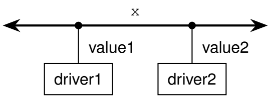
IEEE 1164 std_ulogic Type

- Simple enumeration type with nine values

```
type std_ulogic is ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-');
```

- *Unresolved*: Only supports signals with **single** driver
- Multiple drivers are detected and reported (during elaboration)

```
signal x : std_ulogic;
[...]
driver1 : process(all) is
  [...]
  x <= value1;
end process;
driver2 : process(all) is
  [...]
  x <= value2;
end process;
```




```
[...]: error: too many drivers for signal "x"
```

Special subtype of std_ulogic
subtype std_logic is resolved std_ulogic;

Let us now come to the resolved pendant of the `std_ulogic` type, namely `std_logic`. As shown on the slide, this type is a special subtype of `std_ulogic`.

IEEE 1164 std_logic Type

- Special subtype of `std_ulogic`

```
subtype std_logic is resolved std_ulogic;
```

IEEE SA OPEN

Special subtype of std_ulogic
subtype std_logic is resolved std_ulogic;
std_logic has the same nine values as std_ulogic

Note that the subtype declaration clearly shows that `std_logic` comprises all values of `std_ulogic`.

IEEE 1164 std_logic Type

- Special subtype of `std_ulogic`

```
subtype std_logic is resolved std_ulogic;
```

IEEE SA OPEN

- `std_logic` has the same nine values as `std_ulogic`

■ Special subtype of std_ulogic
subtype std_logic is resolved std_ulogic;
■ std_logic has the same nine values as std_ulogic
■ Allows **multiple** drivers (e.g., wired-AND, tri-state bus, etc.)

As already mentioned before, main difference between std_ulogic and std_logic is how the simulator will handle multiple drivers to a signal of the respective type. Whereas std_ulogic does only permit a single driver and leads to an error whenever this is violated, std_logic allows modelling multiple drivers for the same signal as needed by the wired AND and tri-state examples.

IEEE 1164 std_logic Type

- Special subtype of `std_ulogic`

```
subtype std_logic is resolved std_ulogic;
```

IEEE SA OPEN

- `std_logic` has the same nine values as `std_ulogic`
- Allows **multiple** drivers (e.g., wired-AND, tri-state bus, etc.)

Special subtype of std_ulogic
subtype std_logic is resolved std_ulogic;
std_logic has the same nine values as std_ulogic
Allows **multiple** drivers (e.g., wired-AND, tri-state bus, etc.)
Changing the type of signal x in the previous example does not result in the observed error

Recall the example from the previous slide. If you changed the type of the shared signal `x` to `std_logic`, the simulator would not produce the shown error.

IEEE 1164 std_logic Type

- Special subtype of `std_ulogic`

```
subtype std_logic is resolved std_ulogic;
```

- `std_logic` has the same nine values as `std_ulogic`
- Allows **multiple** drivers (e.g., wired-AND, tri-state bus, etc.)
- Changing the type of signal `x` in the previous example does not result in the observed error

Special subtype of std_ulogic
subtype std_logic is resolved std_ulogic;
std_logic has the same nine values as std_ulogic
Allows **multiple** drivers (e.g., wired-AND, tri-state bus, etc.)
Changing the type of signal x in the previous example does not result in the observed error
There are still multiple drivers ⇒ What value will x exhibit?

However, the issue of multiple drivers obviously still exists. So, how is this handled with `std_logic`? And what value will `x` have in the case of conflicting drivers?

IEEE 1164 std_logic Type

- Special subtype of `std_ulogic`

```
subtype std_logic is resolved std_ulogic;
```

IEEE SA OPEN

- `std_logic` has the same nine values as `std_ulogic`
- Allows **multiple** drivers (e.g., wired-AND, tri-state bus, etc.)
- Changing the type of signal `x` in the previous example does not result in the observed error
  - There are still multiple drivers ⇒ What value will `x` exhibit?

- Special subtype of std_ulogic
  subtype std_logic is resolved std_ulogic;
- std_logic has the same nine values as std_ulogic
- Allows **multiple** drivers (e.g., wired-AND, tri-state bus, etc.)
- Changing the type of signal x in the previous example does not result in the observed error
  - There are still multiple drivers ⇒ What value will x exhibit?
- Uses a *resolution function* (resolved)

The way this is handled is by using a so-called *resolution function*.

In the case of the `std_logic_1164` package this function is called `resolved` and contained in the shown subtype declaration. We will now continue by discussing resolution functions in more detail.

IEEE 1164 std_logic Type

- Special subtype of `std_ulogic`

```
subtype std_logic is resolved std_ulogic;
```

- `std_logic` has the same nine values as `std_ulogic`
- Allows **multiple** drivers (e.g., wired-AND, tri-state bus, etc.)
- Changing the type of signal `x` in the previous example does not result in the observed error
  - There are still multiple drivers ⇒ What value will `x` exhibit?
- Uses a *resolution function* (`resolved`)

Defines resolution of multiple drivers' values into single *resolved value*

A resolution function defines how the conflicting values of multiple drivers to a signal are resolved into a single value. The result of this function is called the *resolved value*.

Resolution Function

- Defines resolution of multiple drivers' values into single *resolved value*

■ Defines resolution of multiple drivers' values into single *resolved value*
■ Pure function featuring single array parameter (all drivers' values)

In general such a resolution function is nothing special. It is merely a *pure* function, which we will introduce in an upcoming lecture, that takes a single parameter. This parameter is an array of the type of the target signal and contains the values of all drivers.

Resolution Function

- Defines resolution of multiple drivers' values into single *resolved value*
- Pure function featuring single array parameter (all drivers' values)

- Defines resolution of multiple drivers' values into single *resolved value*
- Pure function featuring single array parameter (all drivers' values)
- Invoked during the simulation, no real meaning for synthesis

During the simulation, this function is invoked to determine a single, effective, signal value from the multiple driven ones. Note that this has no real meaning for synthesis though, as the resolution of conflicting drivers in real hardware is a result of physics and cannot simply be defined.

Resolution Function

- Defines resolution of multiple drivers' values into single *resolved value*
- Pure function featuring single array parameter (all drivers' values)
- Invoked during the simulation, no real meaning for synthesis

- Defines resolution of multiple drivers' values into single *resolved value*
- Pure function featuring single array parameter (all drivers' values)
- Invoked during the simulation, no real meaning for synthesis
- Resolution functions can be associated to subtypes or signals
  - For subtypes: All signals of this subtype are resolved

In general such resolution functions can be part of a subtype declaration (like you saw for `std_logic`), in which case all signals of the respective type are resolved.

Resolution Function

- Defines resolution of multiple drivers' values into single *resolved value*
- Pure function featuring single array parameter (all drivers' values)
- Invoked during the simulation, no real meaning for synthesis
- Resolution functions can be associated to subtypes or signals
  - For subtypes: All signals of this subtype are resolved

- Defines resolution of multiple drivers' values into single *resolved value*
- Pure function featuring single array parameter (all drivers' values)
- Invoked during the simulation, no real meaning for synthesis
- Resolution functions can be associated to subtypes or signals
  - For subtypes: All signals of this subtype are resolved
  - Arrays and records of subtypes are also supported

Note that also subtypes of arrays and records can be resolved. For details we refer you to the VHDL standard.

Resolution Function

- Defines resolution of multiple drivers' values into single *resolved value*
- Pure function featuring single array parameter (all drivers' values)
- Invoked during the simulation, no real meaning for synthesis
- Resolution functions can be associated to subtypes or signals
  - For subtypes: All signals of this subtype are resolved
  - Arrays and records of subtypes are also supported

44
- Defines resolution of multiple drivers' values into single *resolved value*
- Pure function featuring single array parameter (all drivers' values)
- Invoked during the simulation, no real meaning for synthesis
- Resolution functions can be associated to subtypes or signals
  - For subtypes: All signals of this subtype are resolved
  - Arrays and records of subtypes are also supported
  - For signals: Only respective signal resolved
    Example: `signal x : resolved std_ulogic;`

In addition to that, it is also possible to only resolve a single signal by inserting a resolution function name before the type in a signal declaration. The slide contains an example declaration of a signal `x`, which uses a resolution function called `resolved`.

Resolution Function

- Defines resolution of multiple drivers' values into single *resolved value*
- Pure function featuring single array parameter (all drivers' values)
- Invoked during the simulation, no real meaning for synthesis
- Resolution functions can be associated to subtypes or signals
  - For subtypes: All signals of this subtype are resolved
  - Arrays and records of subtypes are also supported
  - For signals: Only respective signal resolved
    Example: `signal x : resolved std_ulogic;`

Resolves multiple std_ulogic values into a single one

Let us now discuss the the resolution function defined in the `std_logic_1164` package. The implementation of this particular resolution function can be accessed at the linked IEEE repository.

The std_ulogic Resolution Function

- Resolves multiple `std_ulogic` values into a single one IEEE SA OPEN

■ Resolves multiple std_ulogic values into a single one
1 function resolved (s : std_ulogic_vector) return std_ulogic is

While we will only cover functions in an upcoming lecture, the function declaration shown on the slide is simple to grasp, especially if we consider its purpose. In essence, it needs to resolve an array of `std_ulogic` values, called a vector, into a single `std_ulogic` value.

The std_ulogic Resolution Function

- Resolves multiple `std_ulogic` values into a single one IEEE SA OPEN

```
function resolved (s : std_ulogic_vector) return std_ulogic is
```

Resolves multiple std_ulogic values into a single one

```vhdl
function resolved (s : std_ulogic_vector) return std_ulogic is
  variable result : std_ulogic := 'Z';
begin
  if (s'length = 1) then return s(s'low);
  else
    for i in s'range loop
      res := RES_TABLE(result, s(i));
    end loop;
  end if;
  return result;
end function;
```


Next, we can observe a temporary variable being declared which will hold the final resolved value. This variable is initialized to the high impedance value, as this is the weakest driving state.

The std_ulogic Resolution Function

- Resolves multiple `std_ulogic` values into a single one IEEE SA OPEN

```vhdl
function resolved (s : std_ulogic_vector) return std_ulogic is
  variable result : std_ulogic := 'Z';
begin
  if (s'length = 1) then return s(s'low);
  else
    for i in s'range loop
      res := RES_TABLE(result, s(i));
    end loop;
  end if;
  return result;
end function;
```

In case of a single driving value, there is nothing to resolve and the function simply returns this driving value.

The std_ulogic Resolution Function

- Resolves multiple `std_ulogic` values into a single one

```
function resolved (s : std_ulogic_vector) return std_ulogic is
  variable result : std_ulogic := 'Z';
begin
  if (s'length = 1) then return s(s'low);
  else
    for i in s'range loop
      res := RES_TABLE(result, s(i));
    end loop;
  end if;
  return result;
end function;
```

When there are multiple drivers, instead runs over all driving values and iteratively applies a special look-up table, the *resolution table*. Essentially, the resolution is done on consecutive pairs of drivers. We will now discuss this table.

The std_ulogic Resolution Function

- Resolves multiple `std_ulogic` values into a single one

```
function resolved (s : std_ulogic_vector) return std_ulogic is
  variable result : std_ulogic := 'Z';
begin
  if (s'length = 1) then return s(s'low);
  else
    for i in s'range loop
      res := RES_TABLE(result, s(i));
    end loop;
  end if;
  return result;
end function;
```

■ The RES_TABLE defines how two values are resolved into one

As you just heard, the task of the resolution table for `std_ulogic` is to act as a look-up table that provides a resolved value for each pair of `std_ulogic` values.

The std_ulogic Resolution Function (cont'd)

- The `RES_TABLE` defines how two values are resolved into one

On the bottom of the slide, you can find this table as defined in the `std_logic_1164` package. Observe how this is really just a two-dimensional array of `std_ulogic` values. While this table might appear a bit arbitrary, the resolved value for all pairs actually follows some reasoning.

The std_ulogic Resolution Function (cont'd)

- The `RES_TABLE` defines how two values are resolved into one

```
constant RES_TABLE: stdlogic_table := (
-- U    X    0    1    Z    W    L    H    -
--------------------------------------------
 ('U','U','U','U','U','U','U','U','U'), -- U
 ('U','X','X','X','X','X','X','X','X'), -- X
 ('U','X','0','X','0','0','0','0','X'), -- 0
 ('U','X','X','1','1','1','1','1','X'), -- 1
 ('U','X','0','1','Z','W','L','H','X'), -- Z
 ('U','X','0','1','W','W','W','W','X'), -- W
 ('U','X','0','1','L','W','L','W','X'), -- L
 ('U','X','0','1','H','W','W','H','X'), -- H
 ('U','X','X','X','X','X','X','X','X')  -- -
);
```

The RES_TABLE defines how two values are resolved into one

For example, consider the content of the first row, referring to the case where a value must be resolved with an unitialized one. Since the unitialized value can be anything, nothing can be inferred about the outcome. It is hence defined to be unitialized as well. The definition is this way in order for `'U'` values to propagate in the simulation, simplifying it to spot them.

The std_ulogic Resolution Function (cont'd)

- The `RES_TABLE` defines how two values are resolved into one

```
constant RES_TABLE: stdlogic_table := (
-- U    X    0    1    Z    W    L    H    -
-----------------------------------------------
  ('U','U','U','U','U','U','U','U','U'), -- U
  ('U','X','X','X','X','X','X','X','X'), -- X
  ('U','X','0','X','0','0','0','0','X'), -- 0
  ('U','X','X','1','1','1','1','1','X'), -- 1
  ('U','X','0','1','Z','W','L','H','X'), -- Z
  ('U','X','0','1','W','W','W','W','X'), -- W
  ('U','X','0','1','L','W','L','W','X'), -- L
  ('U','X','0','1','H','W','W','H','X'), -- H
  ('U','X','X','X','X','X','X','X','X')  -- -
);
```

The RES_TABLE defines how two values are resolved into one

Now consider the row for the weak unknown value. We already know why the first value entry is `'U'`. Therefore, let's look at the second entry.

The std_ulogic Resolution Function (cont'd)

- The `RES_TABLE` defines how two values are resolved into one

```
constant RES_TABLE: stdlogic_table := (
-- U    X    0    1    Z    W    L    H    -
---------------------------------------------
 ('U','U','U','U','U','U','U','U','U'), -- U
 ('U','X','X','X','X','X','X','X','X'), -- X
 ('U','X','0','X','0','0','0','0','X'), -- 0
 ('U','X','X','1','1','1','1','1','X'), -- 1
 ('U','X','0','1','Z','W','L','H','X'), -- Z
 ('U','X','0','1','W','W','W','W','X'), -- W
 ('U','X','0','1','L','W','L','W','X'), -- L
 ('U','X','0','1','H','W','W','H','X'), -- H
 ('U','X','X','X','X','X','X','X','X')  -- -
);
```

In this case the strong and the weak unknown value must be resolved. Since the strong unknown is associated with a stronger driver strength, it will dominate the weaker driver.

The std_ulogic Resolution Function (cont'd)

- The `RES_TABLE` defines how two values are resolved into one

```
constant RES_TABLE: stdlogic_table := (
-- U    X    0    1    Z    W    L    H    -
------------------------------------------------
 ('U','U','U','U','U','U','U','U','U'), -- U
 ('U','X','X','X','X','X','X','X','X'), -- X
 ('U','X','0','X','0','0','0','0','X'), -- 0
 ('U','X','X','1','1','1','1','1','X'), -- 1
 ('U','X','0','1','Z','W','L','H','X'), -- Z
 ('U','X','0','1','W','W','W','W','X'), -- W
 ('U','X','0','1','L','W','L','W','X'), -- L
 ('U','X','0','1','H','W','W','H','X'), -- H
 ('U','X','X','X','X','X','X','X','X')  -- -
);
```

Conversely, the weak unknown will dominate a high impedance values since this corresponds to the weakest driver strength.

The std_ulogic Resolution Function (cont'd)

- The RES_TABLE defines how two values are resolved into one

```
constant RES_TABLE: stdlogic_table := (
-- U    X    0    1    Z    W    L    H    -
---------------------------------------------
 ('U','U','U','U','U','U','U','U','U'), -- U
 ('U','X','X','X','X','X','X','X','X'), -- X
 ('U','X','0','X','0','0','0','0','X'), -- 0
 ('U','X','X','1','1','1','1','1','X'), -- 1
 ('U','X','0','1','Z','W','L','H','X'), -- Z
 ('U','X','0','1','W','W','W','W','X'), -- W
 ('U','X','0','1','L','W','L','W','X'), -- L
 ('U','X','0','1','H','W','W','H','X'), -- H
 ('U','X','X','X','X','X','X','X','X')  -- -
);
```

To illustrate how multiple driving values are resolved using this table, let us consider an example. For that, recall the initial wired AND example, where we had multiple drivers and a pullup resistor connected to a shared bus. Assume that we have two drivers, one active and one inactive. We can model this using the values `'H'`, `'0'` and `'Z'` being applied to a shared signal.

The std_ulogic Resolution Function (cont'd)

- The `RES_TABLE` defines how two values are resolved into one
  Example: pullup (`'H'`), active (`'0'`) and inactive (`'Z'`) driver

```
constant RES_TABLE: stdlogic_table := (
-- U    X    0    1    Z    W    L    H    -
---------------------------------------------
 ('U','U','U','U','U','U','U','U','U'), -- U
 ('U','X','X','X','X','X','X','X','X'), -- X
 ('U','X','0','X','0','0','0','0','X'), -- 0
 ('U','X','X','1','1','1','1','1','X'), -- 1
 ('U','X','0','1','Z','W','L','H','X'), -- Z
 ('U','X','0','1','W','W','W','W','X'), -- W
 ('U','X','0','1','L','W','L','W','X'), -- L
 ('U','X','0','1','H','W','W','H','X'), -- H
 ('U','X','X','X','X','X','X','X','X')  -- -
);
```

```
resolve("H0Z"):
```

In a first step, the values `'H'` and `'Z'` are resolved. Since `'H'` is associated with a stronger driving strength, it is the return resolved value.

The std_ulogic Resolution Function (cont'd)

- The `RES_TABLE` defines how two values are resolved into one
  Example: pullup (`'H'`), active (`'0'`) and inactive (`'Z'`) driver

```
constant RES_TABLE: stdlogic_table := (
-- U    X    0    1    Z    W    L    H    -
---------------------------------------------
 ('U','U','U','U','U','U','U','U','U'), -- U
 ('U','X','X','X','X','X','X','X','X'), -- X
 ('U','X','0','X','0','0','0','0','X'), -- 0
 ('U','X','X','1','1','1','1','1','X'), -- 1
 ('U','X','0','1','Z','W','L','H','X'), -- Z
 ('U','X','0','1','W','W','W','W','X'), -- W
 ('U','X','0','1','L','W','L','W','X'), -- L
 ('U','X','0','1','H','W','W','H','X'), -- H
 ('U','X','X','X','X','X','X','X','X')  -- -
);
```

```
resolve("H0Z"):
 1  RES_TABLE('H', 'Z') = 'H'
```

Next, this intermediate result must be resolved with `'0'`. The result if of course `'0'`.

The std_ulogic Resolution Function (cont'd)

- The `RES_TABLE` defines how two values are resolved into one
  Example: pullup (`'H'`), active (`'0'`) and inactive (`'Z'`) driver

```
constant RES_TABLE: stdlogic_table := (
-- U    X    0    1    Z    W    L    H    -
---------------------------------------------
 ('U','U','U','U','U','U','U','U','U'), -- U
 ('U','X','X','X','X','X','X','X','X'), -- X
 ('U','X','0','X','0','0','0','0','X'), -- 0
 ('U','X','X','1','1','1','1','1','X'), -- 1
 ('U','X','0','1','Z','W','L','H','X'), -- Z
 ('U','X','0','1','W','W','W','W','X'), -- W
 ('U','X','0','1','L','W','L','W','X'), -- L
 ('U','X','0','1','H','W','W','H','X'), -- H
 ('U','X','X','X','X','X','X','X','X')  -- -
);
```

```
resolve("H0Z"):
1 RES_TABLE('H', 'Z') = 'H'
2 RES_TABLE('H', '0') = '0'
```

Finally, the pair `'0'` and `'Z'` requires resolutions. AS mentioned before, with `'Z'` having the least driving strength, `'0'` will be the result.

The std_ulogic Resolution Function (cont'd)

- The `RES_TABLE` defines how two values are resolved into one
  Example: pullup (`'H'`), active (`'0'`) and inactive (`'Z'`) driver

```
constant RES_TABLE: stdlogic_table := (
-- U   X   0   1   Z   W   L   H   -
---------------------------------------------
 ('U','U','U','U','U','U','U','U','U'), -- U
 ('U','X','X','X','X','X','X','X','X'), -- X
 ('U','X','0','X','0','0','0','0','X'), -- 0
 ('U','X','X','1','1','1','1','1','X'), -- 1
 ('U','X','0','1','Z','W','L','H','X'), -- Z
 ('U','X','0','1','W','W','W','W','X'), -- W
 ('U','X','0','1','L','W','L','W','X'), -- L
 ('U','X','0','1','H','W','W','H','X'), -- H
 ('U','X','X','X','X','X','X','X','X')  -- -
);
```

```
resolve("H0Z"):
1 RES_TABLE('H', 'Z') = 'H'
2 RES_TABLE('H', '0') = '0'
3 RES_TABLE('0', 'Z') = '0'
```

This is also the final result of this driver conflict. Thus, the shared signal will exhibit the value of `'0'`.

The std_ulogic Resolution Function (cont'd)

- The `RES_TABLE` defines how two values are resolved into one
  Example: pullup (`'H'`), active (`'0'`) and inactive (`'Z'`) driver

```
constant RES_TABLE: stdlogic_table := (
-- U    X    0    1    Z    W    L    H    -
---------------------------------------------
 ('U','U','U','U','U','U','U','U','U'), -- U
 ('U','X','X','X','X','X','X','X','X'), -- X
 ('U','X','0','X','0','0','0','0','X'), -- 0
 ('U','X','X','1','1','1','1','1','X'), -- 1
 ('U','X','0','1','Z','W','L','H','X'), -- Z
 ('U','X','0','1','W','W','W','W','X'), -- W
 ('U','X','0','1','L','W','L','W','X'), -- L
 ('U','X','0','1','H','W','W','H','X'), -- H
 ('U','X','X','X','X','X','X','X','X')  -- -
);
```

```
resolve("H0Z"):
 1 RES_TABLE('H', 'Z') = 'H'
 2 RES_TABLE('H', '0') = '0'
 3 RES_TABLE('0', 'Z') = '0'
⇒ resolve("H0Z") = '0'
```

- Common logical operators are defined for `std_ulogic`

In addition to the introduced types, `std_logic_1164` package also provides implementations for some operators on them. Since `std_logic` is a subtype of `std_ulogic` we will limit our considerations to `std_ulogic`.

Logical Operators

- Common logical operators are defined for `std_ulogic`

■ Common logical operators are defined for std_ulogic
■ NOT, AND, OR, XOR, NAND, NOR, XNOR

In particular, the package implements the common logical operators listed on the slide.

Logical Operators

- Common logical operators are defined for `std_ulogic`
  - NOT, AND, OR, XOR, NAND, NOR, XNOR

- Common logical operators are defined for `std_ulogic`
  - NOT, AND, OR, XOR, NAND, NOR, XNOR
- Must respect different semantics of different values

Naturally, the logical operators on `std_ulogic` signals must respect the different semantics of the type's values. For example, consider an AND operation. Regardless of the other input values, a logical low on *any* input, regardless of its driving strength, must always result in a logical low of the output.

Logical Operators

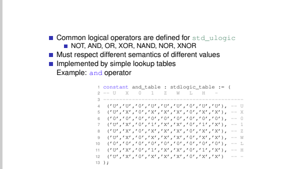
- Common logical operators are defined for `std_ulogic`
  - NOT, AND, OR, XOR, NAND, NOR, XNOR
- Must respect different semantics of different values

■ Common logical operators are defined for std_ulogic
■ NOT, AND, OR, XOR, NAND, NOR, XNOR
■ Must respect different semantics of different values
■ Implemented by simple lookup tables

To express such behavior the implementation provided by the IEEE uses lookup-tables. Similar to the previously discussed resolution table, such operator look-up tables define the operator's outcome for each possible pair of `std_ulogic` input values.

Logical Operators

- Common logical operators are defined for `std_ulogic`
  - NOT, AND, OR, XOR, NAND, NOR, XNOR
- Must respect different semantics of different values
- Implemented by simple lookup tables



The table on the slide shows such a lookup-table for an AND operator.

Logical Operators

- Common logical operators are defined for `std_ulogic`
  - NOT, AND, OR, XOR, NAND, NOR, XNOR
- Must respect different semantics of different values
- Implemented by simple lookup tables
  Example: `and` operator

```
constant and_table : stdlogic_table := (
-- U    X    0    1    Z    W    L    H    -
-------------------------------------------------
  ('U','U','0','U','U','U','0','U','U'), -- U
  ('U','X','0','X','X','X','0','X','X'), -- X
  ('0','0','0','0','0','0','0','0','0'), -- 0
  ('U','X','0','1','X','X','0','1','X'), -- 1
  ('U','X','0','X','X','X','0','X','X'), -- Z
  ('U','X','0','X','X','X','0','X','X'), -- W
  ('0','0','0','0','0','0','0','0','0'), -- L
  ('U','X','0','1','X','X','0','1','X'), -- H
  ('U','X','0','X','X','X','0','X','X')  -- -
);
```

- Common logical operators are defined for std_ulogic
  - NOT, AND, OR, XOR, NAND, NOR, XNOR
- Must respect different semantics of different values
- Implemented by simple lookup tables
  Example: and operator

Observe how a weak low value will **always** result in a strong low result.

Logical Operators

- Common logical operators are defined for `std_ulogic`
  - NOT, AND, OR, XOR, NAND, NOR, XNOR
- Must respect different semantics of different values
- Implemented by simple lookup tables
  Example: `and` operator

```
constant and_table : stdlogic_table := (
-- U    X    0    1    Z    W    L    H    -
---------------------------------------------
  ('U','U','0','U','U','U','0','U','U'), -- U
  ('U','X','0','X','X','X','0','X','X'), -- X
  ('0','0','0','0','0','0','0','0','0'), -- 0
  ('U','X','0','1','X','X','0','1','X'), -- 1
  ('U','X','0','X','X','X','0','X','X'), -- Z
  ('U','X','0','X','X','X','0','X','X'), -- W
  ('0','0','0','0','0','0','0','0','0'), -- L
  ('U','X','0','1','X','X','0','1','X'), -- H
  ('U','X','0','X','X','X','0','X','X')  -- -
);
```

The standard also defines arrays of the new types, called vectors

As we briefly saw before, the IEEE 1164 standard also defines array types for `std_ulogic` and `std_logic`. These array types are referred to as `std_ulogic_vector` respectively `std_logic_vector`.

std_[u]logic_vector Types

- The standard also defines arrays of the new types, called vectors

264
■ The standard also defines arrays of the new types, called vectors
type std_ulogic_vector is array (natural range <>) of std_ulogic;
subtype std_logic_vector is (resolved) std_ulogic_vector;

The slide shows the declaration of these two types. Note how `std_logic_vector` is a resolved type of its unresolved pendant. Since the two vector types are just arrays, we use them in declarations and initialize them just as any other array type. However, instances of these vectors are often used for addresses or data words, therefore often holding numerical values and being quite long.

std_[u]logic_vector Types

- The standard also defines arrays of the new types, called vectors

```
type std_ulogic_vector is array (natural range <>) of std_ulogic;
subtype std_logic_vector is (resolved) std_ulogic_vector;
```

In order to make it easier to assign long, possibly numerical, values to these vectors, VHDL features *bit string literals*.

- The standard also defines arrays of the new types, called vectors
- Vectors can be assigned *bit string literals*

std_[u]logic_vector Types

- The standard also defines arrays of the new types, called vectors

```
type std_ulogic_vector is array (natural range <>) of std_ulogic;
subtype std_logic_vector is (resolved) std_ulogic_vector;
```

- Vectors can be assigned *bit string literals*

The standard also defines arrays of the new types, called vectors
type std_ulogic_vector is array (natural range <>) of std_ulogic;
subtype std_logic_vector is (resolved) std_ulogic_vector;
Vectors can be assigned *bit string literals*
Concise encoding of strings in different numeral systems

Such bit string literals allow it to concisely encode strings in different numeral systems.

std_[u]logic_vector Types

- The standard also defines arrays of the new types, called vectors

```
type std_ulogic_vector is array (natural range <>) of std_ulogic;
subtype std_logic_vector is (resolved) std_ulogic_vector;
```

- Vectors can be assigned *bit string literals*
  - Concise encoding of strings in different numeral systems

The standard also defines arrays of the new types, called vectors
type std_ulogic_vector is array (natural range <>) of std_ulogic;
subtype std_logic_vector is (resolved) std_ulogic_vector;
Vectors can be assigned *bit string literals*
Concise encoding of strings in different numeral systems
bit_string_literal::=[integer]base_specifier"[bit_value]"

On the slide you are provided with the syntax of such a bit string literal. Its characters are limited to the ones of the respective numeral system, plus the ones of `std_ulogic`.

std_[u]logic_vector Types

- The standard also defines arrays of the new types, called vectors

```
type std_ulogic_vector is array (natural range <>) of std_ulogic;
subtype std_logic_vector is (resolved) std_ulogic_vector;
```

- Vectors can be assigned *bit string literals*
  - Concise encoding of strings in different numeral systems

```
bit_string_literal::=[integer]base_specifier"[bit_value]"
```

The standard also defines arrays of the new types, called vectors
type std_ulogic_vector is array (natural range <>) of std_ulogic;
subtype std_logic_vector is (resolved) std_ulogic_vector;
Vectors can be assigned bit string literals
Concise encoding of strings in different numeral systems
bit_string_literal::=[integer]base_specifier"[bit_value]"

The string literal itself, referred to as `bit_value`, is enclosed by double quotation marks and preceded by a `base_specifier`.

std_[u]logic_vector Types

- The standard also defines arrays of the new types, called vectors

```
type std_ulogic_vector is array (natural range <>) of std_ulogic;
subtype std_logic_vector is (resolved) std_ulogic_vector;
```

- Vectors can be assigned *bit string literals*
  - Concise encoding of strings in different numeral systems

```
bit_string_literal::=[integer]base_specifier"[bit_value]"
```

This base specifier defines in which numeral system the bit string is to be interpreted. The specifiers b, x, o and d are used for the binary, hexadecimal, octal, decimal systems, respectively.

std_[u]logic_vector Types

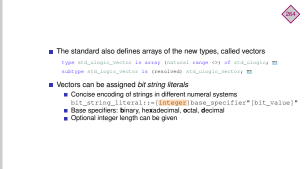
HWMod
WS24

IEEE 1164
Motivation
Standard
VHDL Types
Resolution
Operators
Vector Types
Bit Strings
Operators
Conclusion

- The standard also defines arrays of the new types, called vectors

```
type std_ulogic_vector is array (natural range <>) of std_ulogic;
subtype std_logic_vector is (resolved) std_ulogic_vector;
```

- Vectors can be assigned *bit string literals*
  - Concise encoding of strings in different numeral systems
    `bit_string_literal::=[integer]base_specifier"[bit_value]"`
  - Base specifiers: **b**inary, he**x**adecimal, **o**ctal, **d**ecimal



In general, the string literal corresponding to the bit string must have the same length as the target on the left-hand-side of the assignment. If this is not the case, an error will be raised. However, by providing an optional integer that specifies the target length, it is possible to also give shorter or longer bit strings that are than extended or truncated to fit the target of an assignment. In the case the resulting right-hand side string is shorter than the left-hand side target, the string must be extended. Per default, this extension happens by adding zeros to the left of the string. If the right-hand side string is longer, it must be truncated. This is done by removing leading zeros. If the required truncation to match the length of the left-hand side would lead to non-zero values being truncated, an error will be raised.

std_[u]logic_vector Types

- The standard also defines arrays of the new types, called vectors

```
type std_ulogic_vector is array (natural range <>) of std_ulogic;
subtype std_logic_vector is (resolved) std_ulogic_vector;
```

- Vectors can be assigned *bit string literals*
  - Concise encoding of strings in different numeral systems
    `bit_string_literal::=[integer]base_specifier"[bit_value]"`
  - Base specifiers: **b**inary, he**x**adecimal, **o**ctal, **d**ecimal
  - Optional integer length can be given

The standard also defines arrays of the new types, called vectors
type std_ulogic_vector is array (natural range <>) of std_ulogic;
subtype std_logic_vector is (resolved) std_ulogic_vector;
Vectors can be assigned bit string literals
Concise encoding of strings in different numeral systems
bit_string_literal::=[integer]base_specifier"[bit_value]"
Base specifiers: binary, hexadecimal, octal, decimal
Optional integer length can be given ⇒ "signed" specifiers: sb, sx, so

However, by using the signed base specifiers, indicated by a leading `s`, the extension and truncation consider the left-most bit of the bit string rather than simply zero. Therefore, if a bit string literal that is too short starts with a one, it will be extended by ones. For truncation either only ones or only zeros will be removed, depending on the left-most bit. Let us consider some examples.

std_[u]logic_vector Types

- The standard also defines arrays of the new types, called vectors

```
type std_ulogic_vector is array (natural range <>) of std_ulogic;
subtype std_logic_vector is (resolved) std_ulogic_vector;
```

- Vectors can be assigned *bit string literals*
  - Concise encoding of strings in different numeral systems
    `bit_string_literal::=[integer]base_specifier"[bit_value]"`
  - Base specifiers: **b**inary, he**x**adecimal, **o**ctal, **d**ecimal
  - Optional integer length can be given ⇒ "signed" specifiers: **s**b, **s**x, **s**o

The first example shows a binary bit string comprising eight ones. This is shorter then the target of the assignment. However, since the length is explicitly declared to be eight via the specifier, this bit-string will be extended with zeros accordingly. The comment next to the assignment shows the result. Also observe that the _ sign can be used to format bit strings.

std_[u]logic_vector Types

- The standard also defines arrays of the new types, called vectors

```
type std_ulogic_vector is array (natural range <>) of std_ulogic;
subtype std_logic_vector is (resolved) std_ulogic_vector;
```

- Vectors can be assigned *bit string literals*
  - Concise encoding of strings in different numeral systems
    `bit_string_literal::=[integer]base_specifier"[bit_value]"`
  - Base specifiers: **b**inary, he**x**adecimal, **o**ctal, **d**ecimal
  - Optional integer length can be given ⇒ “signed” specifiers: **s**b, **s**x, **s**o

```
variable u : std_ulogic_vector(7 downto 0) :=  8b"11_1111"; -- 00111111
```

The second example is very similar, but uses a signed base specifier. Since the left-most bit is 1 the resulting value will be extended by 1s as well.

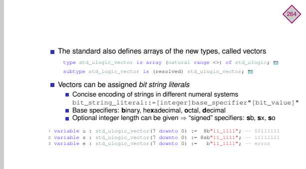
std_[u]logic_vector Types

- The standard also defines arrays of the new types, called vectors

```
type std_ulogic_vector is array (natural range <>) of std_ulogic;
subtype std_logic_vector is (resolved) std_ulogic_vector;
```

- Vectors can be assigned *bit string literals*
  - Concise encoding of strings in different numeral systems
    `bit_string_literal::=[integer]base_specifier"[bit_value]"`
  - Base specifiers: **b**inary, he**x**adecimal, **o**ctal, **d**ecimal
  - Optional integer length can be given ⇒ "signed" specifiers: **s**b, **s**x, **s**o

```
variable u : std_ulogic_vector(7 downto 0) :=  8b"11_1111"; -- 00111111
variable s : std_ulogic_vector(7 downto 0) := 8sb"11_1111"; -- 11111111
```



The third example does not feature a base specifier. Since the bit string literal is too short for the left-hand side this will therefore result in an error during compilation. Keep bit strings in mind, as they are quite handy when initializing long, heterogeneous vectors.

std_[u]logic_vector Types

- The standard also defines arrays of the new types, called vectors

```
type std_ulogic_vector is array (natural range <>) of std_ulogic;
subtype std_logic_vector is (resolved) std_ulogic_vector;
```

- Vectors can be assigned *bit string literals*
  - Concise encoding of strings in different numeral systems
    `bit_string_literal::=[integer]base_specifier"[bit_value]"`
  - Base specifiers: **b**inary, he**x**adecimal, **o**ctal, **d**ecimal
  - Optional integer length can be given ⇒ “signed” specifiers: **s**b, **s**x, **s**o

```
variable u : std_ulogic_vector(7 downto 0) :=  8b"11_1111"; -- 00111111
variable s : std_ulogic_vector(7 downto 0) := 8sb"11_1111"; -- 11111111
variable e : std_ulogic_vector(7 downto 0) :=   b"11_1111"; -- error
```

Finally, the last example shows a bit string literal with hexadecimal base. Observe how only the 3 in the string literal is actually a valid hexadecimal digit, corresponding to the four-bit sequence 0011. However, recall that we said that the string literals can also contain the nine values of the 1164 standard. This is the case for W. Since each digit in a string of hexadecimal string corresponds to four bits, the respective bit sequence is four succeeding Ws.

std_[u]logic_vector Types

- The standard also defines arrays of the new types, called vectors

```
type std_ulogic_vector is array (natural range <>) of std_ulogic;
subtype std_logic_vector is (resolved) std_ulogic_vector;
```

- Vectors can be assigned *bit string literals*
  - Concise encoding of strings in different numeral systems
    `bit_string_literal::=[integer]base_specifier"[bit_value]"`
  - Base specifiers: **b**inary, he**x**adecimal, **o**ctal, **d**ecimal
  - Optional integer length can be given ⇒ “signed” specifiers: **s**b, **s**x, **s**o

```
variable u : std_ulogic_vector(7 downto 0) :=  8b"11_1111"; -- 00111111
variable s : std_ulogic_vector(7 downto 0) := 8sb"11_1111"; -- 11111111
variable e : std_ulogic_vector(7 downto 0) :=   b"11_1111"; -- error
variable h : std_ulogic_vector(7 downto 0) :=   x"3W";      -- 0011WWWW
```

Common logical operators are defined in a bit-wise manner for std_ulogic_vector / std_logic_vector

Just as for the `std_ulogic` type, the `std_logic_1164` package also provides implementations for vector operations. In particular, the package implements the same logical operators as for `std_ulogic` in a bit-wise manner. These bit-wise logical operators take two vector operands of the same length as parameters. They return a vector of identical length as the inputs where each element is determined by performing the logical operation on the respective elements of the parameter vectors.

std_[u]logic_vector Logical Operators

- Common logical operators are defined in a bit-wise manner for `std_ulogic_vector` / `std_logic_vector`

Common logical operators are defined in a bit-wise manner for std_ulogic_vector / std_logic_vector
Example: "UX0011" and "01X0LW" = "0X000X"

For illustration, consider the example of a AND of two `std_ulogic_vector`s on the slide. You can easily determine the result yourself by using the previous AND table.

std_[u]logic_vector Logical Operators

- Common logical operators are defined in a bit-wise manner for `std_ulogic_vector` / `std_logic_vector`
  - Example: `"UX0011" and "01X0LW" = "0X000X"`

- Common logical operators are defined in a bit-wise manner for `std_ulogic_vector` / `std_logic_vector`
  - Example: `"UX0011" and "01X0LW" = "0X000X"`
- Shift operators: `sll, srl`

Furthermore, there are also overloads of four shift operators for `std_ulogic_vector`. The `sll` and `srl` operators are simple logical left, respectively right, shifts They simply shift a vector operand by an integer amount of digits in the respective direction, inserting zeros for the resulting vacant digits.

std_[u]logic_vector Logical Operators

- Common logical operators are defined in a bit-wise manner for `std_ulogic_vector` / `std_logic_vector`
  - Example: `"UX0011" and "01X0LW" = "0X000X"`
- Shift operators: `sll, srl`

- Common logical operators are defined in a bit-wise manner for `std_ulogic_vector` / `std_logic_vector`
  - Example: `"UX0011" and "01X0LW" = "0X000X"`
- Shift operators: `sll, srl`
  - Examples: `"1101" sll 2 = "0100"`, `"1101 srl 2 = "0011"`

Consider the examples on the slide where the vector `1101` is shifted left, respectively right, by two places.

std_[u]logic_vector Logical Operators

- Common logical operators are defined in a bit-wise manner for `std_ulogic_vector` / `std_logic_vector`
  - Example: `"UX0011" and "01X0LW" = "0X000X"`
- Shift operators: `sll, srl`
  - Examples: `"1101" sll 2 = "0100"`, `"1101 srl 2 = "0011"`

- Common logical operators are defined in a bit-wise manner for std_ulogic_vector / std_logic_vector
  - Example: "UX0011" and "01X0LW" = "0X000X"
- Shift operators: sll, srl
  - Examples: "1101" sll 2 = "0100","1101 srl 2 = "0011"
- Rotate operators: rol, ror

The `rol` and `rol` operators, are left and right rotation operators. Here the vacant places at one end of the shifting operation are replaced by the digits moved out at the other end.

std_[u]logic_vector Logical Operators

- Common logical operators are defined in a bit-wise manner for `std_ulogic_vector` / `std_logic_vector`
  - Example: `"UX0011" and "01X0LW" = "0X000X"`
- Shift operators: `sll, srl`
  - Examples: `"1101" sll 2 = "0100"`,`"1101 srl 2 = "0011"`
- Rotate operators: `rol, ror`

Common logical operators are defined in a bit-wise manner for std_ulogic_vector / std_logic_vector
Example: "UX0011" and "01X0LW" = "0X000X"
Shift operators: sll, srl
Examples: "1101" sll 2 = "0100", "1101 srl 2 = "0011"
Rotate operators: rol, ror
Example: "1101" rol 2 = "0111", "1101" ror 2 = "1110"

On the slides these two operations are illustrated via the same `1101` vector from before. Note the difference to the shifting operations.

std_[u]logic_vector Logical Operators

- Common logical operators are defined in a bit-wise manner for `std_ulogic_vector` / `std_logic_vector`
  - Example: `"UX0011" and "01X0LW" = "0X000X"`
- Shift operators: `sll, srl`
  - Examples: `"1101" sll 2 = "0100"`, `"1101 srl 2 = "0011"`
- Rotate operators: `rol, ror`
  - Example: `"1101" rol 2 = "0111"`, `"1101" ror 2 = "1110"`

- Common logical operators are defined in a bit-wise manner for `std_ulogic_vector` / `std_logic_vector`
  - Example: `"UX0011" and "01X0LW" = "0X000X"`
- Shift operators: `sll, srl`
  - Examples: `"1101" sll 2 = "0100"`, `"1101 srl 2 = "0011"`
- Rotate operators: `rol, ror`
  - Example: `"1101" rol 2 = "0111"`, `"1101" ror 2 = "1110"`
- Conversion functions
  - From and to `bit_vector`
  - To differently encoded strings: `to_bstring, to_ostring, to_hstring`

In addition to that, there are also conversion functions to and from the `bit_vector` type, as well as conversion functions to strings encoded with different numerical bases.

std_[u]logic_vector Logical Operators

- Common logical operators are defined in a bit-wise manner for `std_ulogic_vector` / `std_logic_vector`
  - Example: `"UX0011" and "01X0LW" = "0X000X"`
- Shift operators: `sll, srl`
  - Examples: `"1101" sll 2 = "0100"`, `"1101 srl 2 = "0011"`
- Rotate operators: `rol, ror`
  - Example: `"1101" rol 2 = "0111"`, `"1101" ror 2 = "1110"`
- Conversion functions
  - From and to `bit_vector`
  - To differently encoded strings: `to_bstring, to_ostring, to_hstring`

When should which type be used?

Finally, let us end this lecture by a final concluding comparison of the `std_ulogic` and `std_logic` types. In particular, we want to address the question which type should be used when.

std_ulogic vs. std_logic

- When should which type be used?

- When should which type be used?
- Use unresolved types whenever modelled circuit has a single driver

The VHDL standard recommends the use of the unresolved type whenever possible. The resolved type should only be used when the modelled circuit does indeed have multiple drivers.

std_ulogic vs. std_logic

- When should which type be used?
- Use unresolved types whenever modelled circuit has a single driver

When should which type be used?
Use unresolved types whenever modelled circuit has a single driver
Allow tools to detect undesired multiple drivers

This allows the tools to detect and report erroneous multiple drivers, which might lead to undesired behavior.

std_ulogic vs. std_logic

- When should which type be used?
- Use unresolved types whenever modelled circuit has a single driver
  - Allow tools to detect undesired multiple drivers

When should which type be used?
Use unresolved types whenever modelled circuit has a single driver
Allow tools to detect undesired multiple drivers
Most tool generated code and resources use the resolved types

However, virtually all other resources you will find online, and also most tool-generated code, exclusively use `std_logic`. But why? To the best of our knowledge, the reason for that is a historical one.

std_ulogic vs. std_logic

- When should which type be used?
- Use unresolved types whenever modelled circuit has a single driver
  - Allow tools to detect undesired multiple drivers
- Most tool generated code and resources use the resolved types

When should which type be used?
Use unresolved types whenever modelled circuit has a single driver
Allow tools to detect undesired multiple drivers
Most tool generated code and resources use the resolved types
std_logic_vector subtype of std_ulogic_vector since VHDL 2008

The initial version of the 1164 standard did not define `std_logic_vector` as a resolved subtype of `std_ulogic_vector`.

std_ulogic vs. std_logic

- When should which type be used?
- Use unresolved types whenever modelled circuit has a single driver
  - Allow tools to detect undesired multiple drivers
- Most tool generated code and resources use the resolved types
  - `std_logic_vector` subtype of `std_ulogic_vector` since VHDL 2008

When should which type be used?
Use unresolved types whenever modelled circuit has a single driver
Allow tools to detect undesired multiple drivers
Most tool generated code and resources use the resolved types
std_logic_vector subtype of std_ulogic_vector since VHDL 2008
VHDL 1993 required unpleasant type casts

As a result, unpleasant type casts between the vector types were required, leading to designers taking the easy route of simply always using `std_logic_vector`. Since `std_logic` is a resolved subtype of `std_ulogic` that is more permissive regarding multiple drivers, this works fine in most cases. Nevertheless, we do not recommend discarding the safety of the stricter `std_ulogic` type. Since you will learn proper hardware design in this course, we will only use `std_logic` when we **actually** model multiple drivers.

std_ulogic vs. std_logic

HWMod
WS24

IEEE 1164
Motivation
Standard
VHDL Types
Resolution
Operators
Vector Types
Conclusion

- When should which type be used?
- Use unresolved types whenever modelled circuit has a single driver
  - Allow tools to detect undesired multiple drivers
- Most tool generated code and resources use the resolved types
  - `std_logic_vector` subtype of `std_ulogic_vector` since VHDL 2008
  - VHDL 1993 required unpleasant type casts

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.

Lecture Complete!