

At this point you know about the nine-valued logic, the resolution of conflicting drivers, and how this is implemented in the `std_logic_1164` package. In this lecture will pick-up where we left, and introduce the operators and array types provided by the package, and discuss why we even need resolved *and* unresolved types. Furthermore, we will make an excursion to the so-called bit-string literals.

HWMoD
WS25

IEEE 1164
Vectors
IEEE 1164 Recap
Operators
Vector Types
Conclusion

Hardware Modeling [VU] (191.011) – WS25 – Vectors and Bit String Literals (IEEE 1164)

Florian Huemer & Sebastian Wiedemann & Dylan Baumann

WS 2025/26

A key take away of the previous lecture was that Boolean logic cannot express different driver strengths and that it is therefore not able to describe certain circuits. However, sometimes it is necessary to model such circuits when designing digital systems. Hence, a nine-valued logic is used instead of the two-valued Boolean one. This logic contains dedicated values for different driver strengths.

Recall: 9-valued Logic and Resolution Functions

- Boolean logic cannot express different driver strengths

Additionally, to be of use, we also require a resolution function to determine a reasonable outcome for drivers that apply conflicting values to the same wire.

Recall: 9-valued Logic and Resolution Functions

HWMoD
WS25

IEEE 1164
Vectors
IEEE 1164 Recap
Operators
Vector Types
Conclusion

- Boolean logic cannot express different driver strengths
- ⇒ Nine valued logic and resolution of conflicts

For VHDL this logic was standardized in the IEEE 1164 standard and implemented in the `std_logic_1164` package. This package also provides the specific resolution function for the nine-valued logic.

Recall: 9-valued Logic and Resolution Functions

HWMoD
WS25

IEEE 1164
Vectors
IEEE 1164 Recap
Operators
Vector Types
Conclusion

- Boolean logic cannot express different driver strengths
- ⇒ Nine valued logic and resolution of conflicts
- Standardized in IEEE 1164 and implemented in `std_logic_1164` package

The types implementing the logic are the unresolved `std_ulogic` types, and its resolved `std_logic` subtype.

Recall: 9-valued Logic and Resolution Functions

HWMoD
WS25

IEEE 1164
Vectors
IEEE 1164 Recap
Operators
Vector Types
Conclusion

- Boolean logic cannot express different driver strengths
- ⇒ Nine valued logic and resolution of conflicts
- Standardized in IEEE 1164 and implemented in `std_logic_1164` package
 - *Unresolved*: `std_ulogic` IEEE SA [OPEN](#)
 - *Resolved*: `std_logic` IEEE SA [OPEN](#)

Recall that an unresolved type only allows a single driver for a signal, whereas a resolved type allows multiple drivers to assign values to a signal. However, let us now turn our attention to the contents of the package which we did not yet discuss.

Recall: 9-valued Logic and Resolution Functions

HWMMod
WS25

IEEE 1164
Vectors
IEEE 1164 Recap
Operators
Vector Types
Conclusion

- Boolean logic cannot express different driver strengths
- ⇒ Nine valued logic and resolution of conflicts
- Standardized in IEEE 1164 and implemented in `std_logic_1164` package
 - *Unresolved*: `std_ulogic` IEEE SA OPEN Single driver per signal
 - *Resolved*: `std_logic` IEEE SA OPEN Multiple drivers per signal

In particular, we will now continue by

Recall: 9-valued Logic and Resolution Functions

HWMoD
WS25

IEEE 1164
Vectors
IEEE 1164 Recap
Operators
Vector Types
Conclusion

- Boolean logic cannot express different driver strengths
- ⇒ Nine valued logic and resolution of conflicts
- Standardized in IEEE 1164 and implemented in `std_logic_1164` package
 - *Unresolved*: `std_ulogic` IEEE SA [OPEN](#) Single driver per signal
 - *Resolved*: `std_logic` IEEE SA [OPEN](#) Multiple drivers per signal
- What about...

- Boolean logic cannot express different driver strengths
 - ⇒ Nine valued logic and resolution of conflicts
- Standardized in IEEE 1164 and implemented in `std_logic_1164` package
 - *Unresolved*: `std_ulogic` IEEE SA OPEN Single driver per signal
 - *Resolved*: `std_logic` IEEE SA OPEN Multiple drivers per signal
- What about...
 - operations on these types?

looking at the operations the package implements for these types.

Recall: 9-valued Logic and Resolution Functions

HWMMod
WS25

IEEE 1164
Vectors
IEEE 1164 Recap
Operators
Vector Types
Conclusion

- Boolean logic cannot express different driver strengths
- ⇒ Nine valued logic and resolution of conflicts
- Standardized in IEEE 1164 and implemented in `std_logic_1164` package
 - *Unresolved*: `std_ulogic` IEEE SA OPEN Single driver per signal
 - *Resolved*: `std_logic` IEEE SA OPEN Multiple drivers per signal
- What about...
 - operations on these types?

- Boolean logic cannot express different driver strengths
- ⇒ Nine valued logic and resolution of conflicts
- Standardized in IEEE 1164 and implemented in `std_logic_1164` package
 - *Unresolved*: `std_ulogic` IEEE SA OPEN Single driver per signal
 - *Resolved*: `std_logic` IEEE SA OPEN Multiple drivers per signal
- What about...
 - operations on these types?
 - multi-bit signals?

After that we will show you the array types of the `std_ulogic` and `std_logic` types. These are paramount in order to keep your designs concise and maintainable. Just imagine yourself creating thousands of signals for a wide bus with many participants!

Recall: 9-valued Logic and Resolution Functions

- Boolean logic cannot express different driver strengths
- ⇒ Nine valued logic and resolution of conflicts
- Standardized in IEEE 1164 and implemented in `std_logic_1164` package
 - *Unresolved*: `std_ulogic` IEEE SA OPEN Single driver per signal
 - *Resolved*: `std_logic` IEEE SA OPEN Multiple drivers per signal
- What about...
 - operations on these types?
 - multi-bit signals?

- Boolean logic cannot express different driver strengths
 - Nine valued logic and resolution of conflicts
- Standardized in IEEE 1164 and implemented in `std_logic_1164` package
 - *Unresolved*: `std_ulogic` IEEE SA OPEN Single driver per signal
 - *Resolved*: `std_logic` IEEE SA OPEN Multiple drivers per signal
- What about...
 - operations on these types?
 - multi-bit signals?
- And when do you use which?

Finally, we will clarify when you are supposed to use which type. But let's now get to the operators.

Recall: 9-valued Logic and Resolution Functions

- Boolean logic cannot express different driver strengths
- ⇒ Nine valued logic and resolution of conflicts
- Standardized in IEEE 1164 and implemented in `std_logic_1164` package
 - *Unresolved*: `std_ulogic` IEEE SA OPEN Single driver per signal
 - *Resolved*: `std_logic` IEEE SA OPEN Multiple drivers per signal
- What about...
 - operations on these types?
 - multi-bit signals?
- And when do you use which?

Before we look at the them though, be aware that since `std_logic` is a subtype of `std_ulogic` we will limit our considerations to `std_ulogic`.

Logical Operators

HWMoD
WS25

IEEE 1164
Vectors
IEEE 1164 Recap
Operators
Vector Types
Conclusion

The idea behind the `std_ulogic` type is replacing the two-valued `boolean` type for describing logic. Hence, it can hardly come as a surprise that the package implements all common logical operators.

Logical Operators

- Common logical operators are defined for `std_ulogic`

In particular, the operators listed on the slide are defined for `std_ulogic`.

Logical Operators

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

Naturally, these operators must respect the different semantics of the various `std_ulogic` values. For example, consider a logical AND of a weak LOW and a strong HIGH.

Logical Operators

- Common logical operators are defined for `std_ulogic`
 - NOT, AND, OR, XOR, NAND, NOR, XNOR
- Must respect different semantics of different values
 - Example: 'L' and '1' must yield '0'

Regardless of the driving strength of the HIGH values, the result of an AND with *any* LOW must always result in a LOW. In particular, the result is even a strong LOW since a logic gate virtually always has a dedicated output driver.

Logical Operators

- Common logical operators are defined for `std_ulogic`
 - NOT, AND, OR, XOR, NAND, NOR, XNOR
- Must respect different semantics of different values
 - Example: `'1' and '1'` must yield '0'
- Implemented by simple lookup tables

The implementation provided in the `std_logic_1164` package uses lookup-tables to capture the desired result of a function for any pair of `std_ulogic` 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
 - Example: `'1' and '1'` must yield '0'
- Implemented by simple lookup tables

- Common logical operators are defined for `std_ulogic`
 - NOT, AND, OR, XOR, NAND, NOR, XNOR
- Must respect different semantics of different values
 - Example: 'L' and '1' must yield '0'
- Implemented by simple lookup tables

```
Example: and operator
constant and_table : stdlogic_table := (
    0 1 0 1 0 1 0 1
    ('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 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
 - Example: 'L' and '1' must yield '0'
 - Implemented by simple lookup tables
- Example: `and` operator

```
1 constant and_table : stdlogic_table := (
2 -- U X 0 1 Z W L H -
3 -----
4 ('U','U','0','U','U','U','0','U','U'), -- U
5 ('U','X','0','X','X','X','0','X','X'), -- X
6 ('0','0','0','0','0','0','0','0','0'), -- 0
7 ('U','X','0','1','X','X','0','1','X'), -- 1
8 ('U','X','0','X','X','X','0','X','X'), -- Z
9 ('U','X','0','X','X','X','0','X','X'), -- W
10 ('0','0','0','0','0','0','0','0','0'), -- L
11 ('U','X','0','1','X','X','0','1','X'), -- H
12 ('U','X','0','X','X','X','0','X','X') -- -
13 );
```

- Common logical operators are defined for `std_logic`
 - NOT, AND, OR, XOR, NAND, NOR, XNOR
- Must respect different semantics of different values
 - Example: 'L' and '1' must yield '0'
- Implemented by simple lookup tables

```
Example: and operator
constant and_table : stdlogic_table := (
    0 1 0 1 0 1 0 1
    ('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') -- -
);
```

To relate this table to your previous example: Observe how a weak LOW will **always** result in a strong LOW.

Logical Operators

- Common logical operators are defined for `std_ulogic`
 - NOT, AND, OR, XOR, NAND, NOR, XNOR
 - Must respect different semantics of different values
 - Example: 'L' and '1' must yield '0'
 - Implemented by simple lookup tables
- Example: `and` operator

```
1 constant and_table : stdlogic_table := (
2 -- U X 0 1 Z W L H -
3 -----
4 ('U','U','0','U','U','U','0','U','U'), -- U
5 ('U','X','0','X','X','X','0','X','X'), -- X
6 ('0','0','0','0','0','0','0','0','0'), -- 0
7 ('U','X','0','1','X','X','0','1','X'), -- 1
8 ('U','X','0','X','X','X','0','X','X'), -- Z
9 ('U','X','0','X','X','X','0','X','X'), -- W
10 ('0','0','0','0','0','0','0','0','0'), -- L
11 ('U','X','0','1','X','X','0','1','X'), -- H
12 ('U','X','0','X','X','X','0','X','X') -- -
13 );
```

└ Vectors and Bit String Literals (IEEE 1164)

└ Vector Types

└ **std_[u]logic_vector** Types

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

Let's now get to the array types defined in the `std_logic_1164` package, the so-called vectors. There is an dedicated array type for each, `std_ulogic` and `std_logic`.

std_[u]logic_vector Types

264

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

HWMod
WS25

IEEE 1164
Vectors
IEEE 1164 Recap
Operators
Vector Types
Bit Strings
Operators
Conclusion

└ Vectors and Bit String Literals (IEEE 1164)

└ Vector Types

└ **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; IEEE SA
subtype std_logic_vector is (resolved) std_ulogic_vector; IEEE SA

They are fittingly called `std_ulogic_vector` and `std_logic_vector` and you can find their declarations on the slide.

std_[u]logic_vector Types

264

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

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

HWMoD
WS25

IEEE 1164
Vectors
IEEE 1164 Recap
Operators
Vector Types
Bit Strings
Operators
Conclusion

└ Vectors and Bit String Literals (IEEE 1164)

└ Vector Types

└ **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; IEEE 1164
subtype std_logic_vector is (resolved) std_ulogic_vector; IEEE 1164

Note how `std_logic_vector` is a resolved type of its unresolved pendant.

std_[u]logic_vector Types

264

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

```
type std_ulogic_vector is array (natural range <>) of std_ulogic; IEEE 1164  
subtype std_logic_vector is (resolved) std_ulogic_vector; IEEE 1164
```

HWMoD
WS25

IEEE 1164
Vectors
IEEE 1164 Recap
Operators
Vector Types
Bit Strings
Operators
Conclusion

└ Vectors and Bit String Literals (IEEE 1164)

└ Vector Types

└ **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; IEEE SA
std_logic_vector is resolved std_ulogic_vector; IEEE SA

Since the two vector types are just arrays, we can use them in declarations and initialize them just as any other array. However, instances of these vectors are often used for quite long address or data words that hold a numerical value. Thus, simply assigning them a value as you saw in a previous lecture can be quite tedious and error-prone.

std_[u]logic_vector Types

264

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

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

HWMoD
WS25

IEEE 1164
Vectors
IEEE 1164 Recap
Operators
Vector Types
Bit Strings
Operators
Conclusion

Vectors and Bit String Literals (IEEE 1164)

Vector Types

`std_[u]logic_vector` Types

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

To address this, VHDL provides *bit string literals* that can be assigned to vectors.

`std_[u]logic_vector` Types

264

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

```
type std_ulogic_vector is array (natural range <>) of std_ulogic; IEEE 1164  
subtype std_logic_vector is (resolved) std_ulogic_vector; IEEE 1164
```

- Vectors can be assigned *bit string literals*

HWMoD
WS25

IEEE 1164
Vectors
IEEE 1164 Recap
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

These special string literals allow it to concisely encode vector values in different numeral systems.

`std_[u]logic_vector` Types

264

HWMoD
WS25

IEEE 1164
Vectors
IEEE 1164 Recap
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; IEEE SA  
OPEN  
subtype std_logic_vector is (resolved) std_ulogic_vector; IEEE SA  
OPEN
```

- 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]"`

You can find the syntax of such a bit string literal on the slide.

`std_[u]logic_vector` Types

264

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

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

- Vectors can be assigned *bit string literals*

- Concise encoding of strings in different numeral systems

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

The actual string literal is the part enclosed in double quotation marks and referred to as `bit_value`. It is essentially just a sequence of characters that either belong to the used numerical system, or to the nine-valued logic. For example, for the decimal system it would be a sequence of the characters 0 to 9, as well as the nine values of `std_ulogic`.

`std_[u]logic_vector` Types

264

HWMoD
WS25

IEEE 1164
Vectors
IEEE 1164 Recap
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; IEEE 1164  
subtype std_logic_vector is (resolved) std_ulogic_vector; IEEE 1164
```

- 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 `bit_value` is preceded by a part called `base_specifier` that defines in which numeral system the bit string is to be interpreted.

`std_[u]logic_vector` Types

264

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

```
type std_ulogic_vector is array (natural range <>) of std_ulogic; IEEE 1164  
subtype std_logic_vector is (resolved) std_ulogic_vector; IEEE 1164
```

- Vectors can be assigned *bit string literals*

- Concise encoding of strings in different numeral systems

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

The specifiers **b**, **x**, **o** and **d** are used for the binary, hexadecimal, octal, and decimal systems, respectively.

std_[u]logic_vector Types

264

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

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

- 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**

Vectors and Bit String Literals (IEEE 1164)

Vector Types

`std_[u]logic_vector` Types

- The standard also defines arrays of the new types, called vectors
- Vectors can be assigned *bit string literals*
- Concise encoding of strings in different numeral systems
- Base specifiers: **binary**, **hexadecimal**, **octal**, **decimal**

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.

`std_[u]logic_vector` Types

264

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

```
type std_ulogic_vector is array (natural range <>) of std_ulogic; IEEE 1164  
subtype std_logic_vector is (resolved) std_ulogic_vector; IEEE 1164
```

- 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**

HWMoD
WS25

IEEE 1164
Vectors
IEEE 1164 Recap
Operators
Vector Types
Bit Strings
Operators
Conclusion

Vectors and Bit String Literals (IEEE 1164)

Vector Types

`std_[u]logic_vector` Types

- The standard also defines arrays of the new types, called vectors
- Vectors can be assigned *bit string literals*
- Concise encoding of strings in different numeral systems
- Base specifiers: **binary**, **hexadecimal**, **octal**, **decimal**
- Optional integer length can be given

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 the assignment.

`std_[u]logic_vector` Types

264

HWMoD
WS25

IEEE 1164
Vectors
IEEE 1164 Recap
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; IEEE 1164  
subtype std_logic_vector is (resolved) std_ulogic_vector; IEEE 1164
```

- 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

Vectors and Bit String Literals (IEEE 1164)

Vector Types

`std_[u]logic_vector` Types

- The standard also defines arrays of the new types, called vectors
- Vectors can be assigned *bit string literals*
- Concise encoding of strings in different numeral systems
- Base specifiers: **binary**, **hexadecimal**, **octal**, **decimal**
- Optional integer length can be given

In the case of the resulting right-hand side string being shorter than the left-hand side target, the string must be extended. Per default, this happens by adding zeros left of the given string.

`std_[u]logic_vector` Types

264

HWMoD
WS25

IEEE 1164
Vectors
IEEE 1164 Recap
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; IEEE 1164  
subtype std_logic_vector is (resolved) std_ulogic_vector; IEEE 1164
```

- 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

Vectors and Bit String Literals (IEEE 1164)

Vector Types

`std_[u]logic_vector` Types

- The standard also defines arrays of the new types, called vectors
- Vectors can be assigned *bit string literals*
 - Concise encoding of strings in different numeral systems
 - Base specifiers: **binary**, **hexadecimal**, **octal**, **decimal**
 - Optional integer length can be given

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

264

HWMoD
WS25

IEEE 1164
Vectors
IEEE 1164 Recap
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; IEEE 1164  
subtype std_logic_vector is (resolved) std_ulogic_vector; IEEE 1164
```

- 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

Vectors and Bit String Literals (IEEE 1164)

Vector Types

std_[u]logic_vector Types

- The standard also defines arrays of the new types, called vectors
- Vectors can be assigned bit string literals
- Concise encoding of strings in different numeral systems
- 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.

std_[u]logic_vector Types

264

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

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

- 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**

HWMoD
WS25

IEEE 1164
Vectors
IEEE 1164 Recap
Operators
Vector Types
Bit Strings
Operators
Conclusion

└ Vectors and Bit String Literals (IEEE 1164)

└ Vector Types

└ `std_[u]logic_vector` Types

- The standard also defines arrays of the new types, called vectors
- Vectors can be assigned *bit string literals*
- Concise encoding of strings in different numeral systems
- Base specifiers: **binary**, **hexadecimal**, **octal**, **decimal**
- Optional integer length can be given ⇒ “signed” specifiers: **sb**, **sx**, **so**

That’s been quite a lot information! Let us now consider some examples.

std_[u]logic_vector Types

264

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

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

- 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**

HWMoD
WS25

IEEE 1164
Vectors
IEEE 1164 Recap
Operators
Vector Types
Bit Strings
Operators
Conclusion

- The standard also defines arrays of the new types, called vectors
- Vectors can be assigned *bit string literals*
 - Concise encoding of strings in different numeral systems
 - Base specifiers: **binary**, **hexadecimal**, **octal**, **decimal**
 - Optional integer length can be given ⇒ “signed” specifiers: **sb**, **sx**, **so**

Our first example is a binary bit string comprising six ones. This is obviously shorter than 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

264

HWMoD
WS25

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

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

- 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**

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

IEEE 1164
Vectors
IEEE 1164 Recap
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 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**

```
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 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.

std_[u]logic_vector Types

264

HWMoD
WS25

IEEE 1164
Vectors
IEEE 1164 Recap
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; IEEE SA
subtype std_logic_vector is (resolved) std_ulogic_vector; OPEN
```

- 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**

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

- 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 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**

```
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
```

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.

std_[u]logic_vector Types

264

HWMoD
WS25

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

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

- 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**

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

IEEE 1164
Vectors
IEEE 1164 Recap
Operators
Vector Types
Bit Strings
Operators
Conclusion

- The standard also defines arrays of the new types, called vectors
- Vectors can be assigned bit string literals
 - Concise encoding of strings in different numeral systems
 - Base specifiers: binary, hexadecimal, octal, decimal
 - Optional integer length can be given ⇒ "signed" specifiers: **sb**, **sx**, **so**

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. It encodes 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'. And since each digit in a hexadecimal string corresponds to four bits, the bit sequence matching the character Ws is four succeeding weak UNKNOWNs.

std_[u]logic_vector Types

264

HWMoD
WS25

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

```
type std_ulogic_vector is array (natural range <>) of std_ulogic; IEEE SA 1164-2001
subtype std_logic_vector is (resolved) std_ulogic_vector; IEEE SA 1164-2001
```

- 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**

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

Vectors and Bit String Literals (IEEE 1164)

Vector Types

std_[u]logic_vector Types

- The standard also defines arrays of the new types, called vectors
- Vectors can be assigned *bit string literals*
 - Concise encoding of strings in different numeral systems
 - Base specifiers: **binary**, **hexadecimal**, **octal**, **decimal**
 - Optional integer length can be given ⇒ “signed” specifiers: **sb**, **sx**, **so**

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

Try to keep bit strings in mind. They can come in quite handy when initializing long, heterogeneous vectors.

std_[u]logic_vector Types

264

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

```
type std_ulogic_vector is array (natural range <>) of std_ulogic; IEEE 1164
subtype std_logic_vector is (resolved) std_ulogic_vector; IEEE 1164
```

- 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**

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

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.

std_[u]logic_vector Logical Operators

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

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`
 - Length of both arguments and the return value are equal

For illustration, consider the example AND of two `std_u_logic_vectors` on the slide. Each bit of the result is the logical AND of the respective input bits. For example, the first bit of the result is '0' and '0' and thus '0'. You can easily confirm the other bits of the shown 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_u_logic_vector` / `std_logic_vector`
 - Length of both arguments and the return value are equal
 - Example: `"UX0011"` and `"01X01W"` = `"0X000X"`

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 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`
 - Length of both arguments and the return value are equal
 - Example: `"UX0011"` and `"01X0LW"` = `"0X000X"`
- Shift operators: `sll`, `srl`

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_u_logic_vector` / `std_logic_vector`
 - Length of both arguments and the return value are equal
 - Example: `"UX0011"` and `"01X0LW"` = `"0X000X"`
- Shift operators: `sll`, `srl`
 - Examples: `"1101" sll 2 = "0100"`, `"1101" srl 2 = "0011"`

The `rol` and `ror` 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`
 - Length of both arguments and the return value are equal
 - 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`
 - Length of both arguments and the return value are equal
- Example: `"0101" and "1001" = "0001"`
- Shift operators: `sll`, `srl`
 - Example: `"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 by rotating the same `1101` vector from before. Note the difference to the shifting operations. In particular, the amount of zeros and ones did not change by performing a rotation.

`std_[u]logic_vector` Logical Operators

- Common logical operators are defined in a bit-wise manner for `std_ulogic_vector` / `std_logic_vector`
 - Length of both arguments and the return value are equal
 - 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
■ Length of both arguments and the return value are equal
■ Example: "0011" and "1001" = "0001"
■ Shift operators: sll, srl
■ Example: "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`
 - Length of both arguments and the return value are equal
 - 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`

- Common logical operators are defined in a bit-wise manner for `std_ulogic_vector` / `std_logic_vector`
 - Length of both arguments and the return value are equal
 - Example: `"0011" and "1001" = "0001"`
- Shift operators: `sll`, `srl`
 - Example: `"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`

With that we are done with introducing the functionality of the `std_logic_1164` package.

`std_[u]logic_vector` Logical Operators

- Common logical operators are defined in a bit-wise manner for `std_ulogic_vector` / `std_logic_vector`
 - Length of both arguments and the return value are equal
 - 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`

This only leaves one question open: When should you use an unresolved type, and when its resolved subtype?

std_ulogic vs. std_logic

564

HWMoD
WS25

IEEE 1164
Vectors
IEEE 1164 Recap
Operators
Vector Types
Conclusion

- When should which type be used?

The VHDL standard has a direct recommendation regarding this question: use 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

564

HWMoD
WS25

IEEE 1164
Vectors
IEEE 1164 Recap
Operators
Vector Types
Conclusion

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

This allows the tools to detect and report erroneous multiple drivers, which might lead to undesired behavior. However, why even address this question if it is obvious?

std_ulogic vs. std_logic

564

HWMoD
WS25

IEEE 1164
Vectors
IEEE 1164 Recap
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

- 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

The reason is that virtually all other resources you will find online, and also most tool-generated code, exclusively use the resolved `std_logic` subtype. But why would they do that if the standard recommends otherwise?

std_ulogic vs. std_logic

564

HWMoD
WS25

IEEE 1164
Vectors
IEEE 1164 Recap
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

To the best of our knowledge, the reason for that is a purely historical one.

std_ulogic vs. std_logic

564

HWMoD
WS25

IEEE 1164
Vectors
IEEE 1164 Recap
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

- 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
 - VHDL 1993 required unpleasant type casts

The initial version of the 1164 standard did not define `std_logic_vector` as a resolved subtype of `std_ulogic_vector`. As a result, unpleasant type casts between the vector types were required, leading to designers taking the easy route of always using `std_logic_vector`. And consequently also `std_logic` was used instead of `std_ulogic`. Usually this practice works just fine as `std_logic` provides the same values as `std_ulogic`.

`std_ulogic` vs. `std_logic`

564

HWMMod
WS25

IEEE 1164
Vectors
IEEE 1164 Recap
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
 - VHDL 1993 required unpleasant type casts

- 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
 - VHDL 1993 required unpleasant type casts
 - Just using `std_logic_vector` hides undesired multiple drivers

However, in case you erroneously model multiple drivers for some signal, exclusively using the resolved types will hide this and can lead to deviations of the hardware from its expected behavior that are hard to locate. So what will you do in practice? Will you prefer the ugly type casts or the masking of driver conflicts?

std_ulogic vs. std_logic

564

HWMoD
WS25

IEEE 1164
Vectors
IEEE 1164 Recap
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
 - VHDL 1993 required unpleasant type casts
 - Just using `std_logic_vector` hides 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
 - VHDL 1993 required unpleasant type casts
 - Just using `std_logic_vector` hides undesired multiple drivers
 - `std_logic_vector` subtype of `std_ulogic_vector` since VHDL 2008

Well, since VHDL 2008 there is no need to decide. Since this version of VHDL `std_logic_vector` is a proper resolved subtype of `std_ulogic_vector`. You could already observe this earlier when we showed you the declarations of the two vector types. Therefore, you should embrace the safety of the stricter `std_ulogic` type and only use `std_logic` when you **actually** model multiple drivers.

std_ulogic vs. std_logic

564

HWMoD
WS25

IEEE 1164
Vectors
IEEE 1164 Recap
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
 - VHDL 1993 required unpleasant type casts
 - Just using `std_logic_vector` hides undesired multiple drivers
 - `std_logic_vector` subtype of `std_ulogic_vector` since VHDL 2008

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

HWMMod
WS25

IEEE 1164
Vectors
IEEE 1164 Recap
Operators
Vector Types
Conclusion

Lecture Complete!