Hardware Modeling [VU] (191.011)
– WS25 –
Numeric Standard Package

Florian Huemer & Sebastian Wiedemann & Dylan Baumann

WS 2025/26

In the previous lecture we discussed the limits of Boolean types for describing hardware and introduced the `std_ulogic` and its resolved `std_logic` type as a remedy. However, similar limits also exist for the types used for integer arithmetic. This is what we will be concerned with in this lecture. After you watched this video, you can recall the types defined in the numeric-standard package, explain how they differ from the integer type and how you can perform conversions between the numeric-standard, IEEE 1164 and integer types.

numeric_std
Motivation
IEEE Package
Types
Operators
Conversion

Hardware Modeling [VU] (191.011)
– WS25 –
Numeric Standard Package

Florian Huemer & Sebastian Wiedemann & Dylan Baumann

WS 2025/26

Modified: 2025-12-16, 16:06 (f8a58e9)

■ When possible use ranged `integer` for synthesizable code

When it comes to integer arithmetic, we already introduced the integer type, as well as its ranged subtypes. When you write code for synthesis, we recommend you to use such ranged integers whenever possible, as they are concise and easily comprehensible.

Integer Arithmetic in Hardware

- When possible use ranged `integer` for synthesizable code

- When possible use ranged `integer` for synthesizable code
- However: `integer` type has some limitations

However, similar to the Boolean type being inadequate for describing some hardware, the integer type has some limitations that restrict it from being used to model all integer arithmetic in hardware.

Integer Arithmetic in Hardware

- When possible use ranged `integer` for synthesizable code
- However: `integer` type has some limitations

- When possible use ranged `integer` for synthesizable code
- However: `integer` type has some limitations
  - Only 32 / 64 bits guaranteed by standard (2008 / 2019)

A major problem is the maximum range of the integer type. While the VHDL standard does not explicitly define the number of bits of an integer, it is required to be at least 32 or 64 bit depending on the version of the standard. In practice this limits the integer arithmetic we model to a fixed range as well, since we cannot rely upon all and platforms supporting more. However, there is no reason why the hardware we design should be limited to this rather arbitrary range, and it is not unusual that numbers not fitting this range are required.

Integer Arithmetic in Hardware

numeric_std
Motivation
IEEE Package
Types
Operators
Conversion

- When possible use ranged `integer` for synthesizable code
- However: `integer` type has some limitations
  - Only 32 / 64 bits guaranteed by standard (2008 / 2019)

- When possible use ranged `integer` for synthesizable code
- However: `integer` type has some limitations
  - Only 32 / 64 bits guaranteed by standard (2008 / 2019)
  - Restricted to Boolean 0 and 1 vs. 9-valued logic

Furthermore, the integer type is restricted to proper binary numbers. It would be beneficial for debugging though, if the numeric types can also take the nine values of the IEEE 1164 standard.

Integer Arithmetic in Hardware

- When possible use ranged `integer` for synthesizable code
- However: `integer` type has some limitations
  - Only 32 / 64 bits guaranteed by standard (2008 / 2019)
  - Restricted to Boolean 0 and 1 vs. 9-valued logic

- When possible use ranged integer for synthesizable code
- However: integer type has some limitations
  - Only 32 / 64 bits guaranteed by standard (2008 / 2019)
  - Restricted to Boolean 0 and 1 vs. 9-valued logic
  - Behavior on over-/underflow undefined ⇒ wrap-around **or** runtime error

Finally, the behavior of an integer over- or underflow during simulation is undefined and hence depends on the respective simulator's implementation. It should be pointed out though, that this is only the case for integers without user-declared ranges. For range-constrained integer subtypes an over- or underflow will always lead to an error during the simulation. For the unconstrained integer type some simulators will wrap-around on such an event, whereas other simulators will generate a runtime error. Naturally, synthesized hardware cannot simply raise an error and instead values wrap-around on an over- or underflow. As we would like our simulation to behave as similar to the modelled hardware as possible, we ideally require a numerical type that wraps around in simulation.

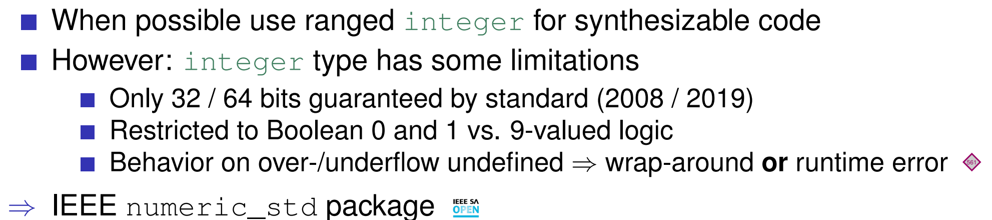
Integer Arithmetic in Hardware

- When possible use ranged `integer` for synthesizable code
- However: `integer` type has some limitations
  - Only 32 / 64 bits guaranteed by standard (2008 / 2019)
  - Restricted to Boolean 0 and 1 vs. 9-valued logic
  - Behavior on over-/underflow undefined ⇒ wrap-around **or** runtime error

When possible use ranged integer for synthesizable code
However: integer type has some limitations
Only 32 / 64 bits guaranteed by standard (2008 / 2019)
Restricted to Boolean 0 and 1 vs. 9-valued logic
Behavior on over-/underflow undefined ⇒ wrap-around **or** runtime error
⇒ IEEE numeric_std package

To cope with the limitations of the integer type for describing arithmetic operations in hardware, the IEEE defines the `numeric_std` package. This package provides additional types and corresponding operators, which we will discuss next.

Integer Arithmetic in Hardware

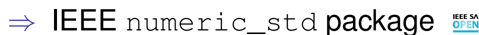


- When possible use ranged `integer` for synthesizable code
- However: `integer` type has some limitations
  - Only 32 / 64 bits guaranteed by standard (2008 / 2019)
  - Restricted to Boolean 0 and 1 vs. 9-valued logic
  - Behavior on over-/underflow undefined ⇒ wrap-around **or** runtime error
⇒ IEEE `numeric_std` package
- Requires import from the `ieee` library
library ieee;
use ieee.numeric_std.all;

Note that the `numeric_std` package must be imported just like the `std_logic_1164` package. However, in case you want to import multiple packages from the IEEE library, the library statement is only required once.

Integer Arithmetic in Hardware

- When possible use ranged `integer` for synthesizable code
- However: `integer` type has some limitations
  - Only 32 / 64 bits guaranteed by standard (2008 / 2019)
  - Restricted to Boolean 0 and 1 vs. 9-valued logic
  - Behavior on over-/underflow undefined ⇒ wrap-around **or** runtime error



- Requires import from the `ieee` library

```
library ieee;
use ieee.numeric_std.all;
```

unsigned and signed with respective operator definitions

The IEEE `numeric_std` package provides two new types, called `unsigned` and `signed`, as well as operators on these types. We will discuss both the types and the operators in detail on the next slide. First though, we want to illustrate the different overflow, respectively underflow, behavior of `integer` and the new types.

The IEEE numeric_std Package

HWMod
WS25

numeric_std
Motivation
IEEE Package
Types
Operators
Conversion

- `unsigned` and `signed` with respective operator definitions

Consider the code snippet shown on the slide that increments a variable by 1, reporting its value before and after incrementing.

The IEEE numeric_std Package

HWMod WS25
numeric_std
Motivation
IEEE Package
Types
Operators
Conversion

- unsigned and signed with respective operator definitions
- Wrap-around on over-/underflow behavior

```
1 process is
2   variable a : integer
3     := integer'high;
4 begin
5   report to_string(a);
6   a := a + 1;
7   report to_string(a);
8   wait;
9 end process;
```

■ unsigned and signed with respective operator definitions
■ Wrap-around on over-/underflow behavior

The variable is of type `integer` and initialized to the highest possible value an object of this type can hold.

The IEEE numeric_std Package

HWMod
WS25

numeric_std
Motivation
IEEE Package
Types
Operators
Conversion

- `unsigned` and `signed` with respective operator definitions
- Wrap-around on over-/underflow behavior

```
1 process is
2   variable a : integer
3     := integer'high;
4 begin
5   report to_string(a);
6   a := a + 1;
7   report to_string(a);
8   wait;
9 end process;
```

unsigned and signed with respective operator definitions
Wrap-around on over-/underflow behavior

The result of the first `report` statement is this value, shown on the slide. This basically shows us that the used simulator uses 32-bit for integers, since this is the greatest number a 32-bit integer can hold. What will now happen when the variable holding this value is incremented by one?

The IEEE numeric_std Package

- `unsigned` and `signed` with respective operator definitions
- Wrap-around on over-/underflow behavior

```
1 process is
2   variable a : integer
3     := integer'high;
4 begin
5   report to_string(a);
6   a := a + 1;
7   report to_string(a);
8   wait;
9 end process;
```

```
[...]: 2147483647
```

- unsigned and signed with respective operator definitions
- Wrap-around on over-/underflow behavior

```
1 process is
2   variable a : integer
3     := integer'high;
4 begin
5   report to_string(a);
6   a := a + 1;
7   report to_string(a);
8   wait;
9 end process;
```

```
[...]: 2147483647
[...]: error: overflow detected
```

As hinted at before, the simulator will detect the overflow and produce an error output as the one shown on the slide.

The IEEE numeric_std Package

HWMod
WS25

numeric_std
Motivation
IEEE Package
Types
Operators
Conversion

- unsigned and signed with respective operator definitions
- Wrap-around on over-/underflow behavior

```
1 process is
2   variable a : integer
3     := integer'high;
4 begin
5   report to_string(a);
6   a := a + 1;
7   report to_string(a);
8   wait;
9 end process;
```

```
[...]: 2147483647
[...]: error: overflow detected
```

unsigned and signed with respective operator definitions
Wrap-around on over-/underflow behavior

Let us now consider the code snippet shown to the right of the first one.

The IEEE numeric_std Package

numeric_std
Motivation
IEEE Package
Types
Operators
Conversion

- unsigned and signed with respective operator definitions
- Wrap-around on over-/underflow behavior

```
1 process is
2   variable a : integer
3     := integer'high;
4 begin
5   report to_string(a);
6   a := a + 1;
7   report to_string(a);
8   wait;
9 end process;
```

```
1 process is
2   variable a : signed(31 downto 0)
3     := to_signed(integer'high, 32);
4 begin
5   report to_string(to_integer(a));
6   a := a + 1;
7   report to_string(to_integer(a));
8   wait;
9 end process;
```

```
[...]: 2147483647
[...]: error: overflow detected
```

It basically does the same as the previous code with the main difference being that the variable is now of type `signed` rather than `integer`. Since we determined the `integer` type in our simulator being 32-bits wide, we also use 32-bits now. Furthermore, we again initialize this variable to the maximum possible integer value. However, note that this time we use a conversion function for that. We will elaborate on this shortly, but let us now continue our discussion of the overflow behavior.

The IEEE numeric_std Package

- `unsigned` and `signed` with respective operator definitions
- Wrap-around on over-/underflow behavior

```
1 process is
2   variable a : integer
3     := integer'high;
4 begin
5   report to_string(a);
6   a := a + 1;
7   report to_string(a);
8   wait;
9 end process;
```

```
1 process is
2   variable a : signed(31 downto 0)
3     := to_signed(integer'high, 32);
4 begin
5   report to_string(to_integer(a));
6   a := a + 1;
7   report to_string(to_integer(a));
8   wait;
9 end process;
```

```
[...]: 2147483647
[...]: error: overflow detected
```

The first `report` statement yields the exact same result as the one in the previous code. This should not be much of a surprise. But what about the second one? What will be the result of incrementing `a`?

The IEEE numeric_std Package

- `unsigned` and `signed` with respective operator definitions
- Wrap-around on over-/underflow behavior

```
1 process is
2   variable a : integer
3     := integer'high;
4 begin
5   report to_string(a);
6   a := a + 1;
7   report to_string(a);
8   wait;
9 end process;
```

```
[...]: 2147483647
[...]: error: overflow detected
```

```
1 process is
2   variable a : signed(31 downto 0)
3     := to_signed(integer'high, 32);
4 begin
5   report to_string(to_integer(a));
6   a := a + 1;
7   report to_string(to_integer(a));
8   wait;
9 end process;
```

```
[...]: 2147483647
```

unsigned and signed with respective operator definitions
Wrap-around on over-/underflow behavior

As already mentioned, for the `signed` type the overflow will wrap around. As a result, we can observe the smallest possible decimal one can store in 32-bits.

The IEEE numeric_std Package

- `unsigned` and `signed` with respective operator definitions
- Wrap-around on over-/underflow behavior

```
1 process is
2   variable a : integer
3     := integer'high;
4 begin
5   report to_string(a);
6   a := a + 1;
7   report to_string(a);
8   wait;
9 end process;
```

```
[...]: 2147483647
[...]: error: overflow detected
```

```
1 process is
2   variable a : signed(31 downto 0)
3     := to_signed(integer'high, 32);
4 begin
5   report to_string(to_integer(a));
6   a := a + 1;
7   report to_string(to_integer(a));
8   wait;
9 end process;
```

```
[...]: 2147483647
[...]: -2147483648
```

Resolved array types of std_ulogic

Alright, let us now properly introduce the two types provided by the `numeric_std` package. In essence, both new types are just resolved `std_ulogic` arrays.

The unsigned and signed Types

numeric_std
Motivation
IEEE Package
Types
Operators
Conversion

- Resolved array types of `std_ulogic`

Resolved array types of std_ulogic
Can be interpreted as std_logic_vector for arithmetic

We can thus simply interpret them as dedicated `std_logic_vector` for arithmetic. At this point you might ask yourself why we even need new types if they are essentially just the ones we already discussed before. However, when carrying out arithmetic operations, the basic `std_ulogic_vector` and `std_logic_vector` types lack something paramount: An *interpretation* of their elements. To elaborate on what we mean by that, try to answer the following question: What numerical number does a certain binary string represent? The thins is, depending on the used encoding, it could be signed, unsigned, encoded in two's complement or unary et cetera. There really is no way of simple disambiguating between different interpretations just given a binary string alone.

The unsigned and signed Types

numeric_std
Motivation
IEEE Package
Types
Operators
Conversion

- Resolved array types of `std_ulogic`
  - Can be interpreted as `std_logic_vector` for arithmetic

Resolved array types of std_ulogic
Can be interpreted as std_logic_vector for arithmetic
Represent unsigned / two's complement binary integers

This is where the `unsigned` and `signed` types come in, as their purpose is to define how the data contained in an array of `std_ulogic` or `std_logic` must be interpreted. In particular, the `unsigned` type is simply interpreted as an unsigned, binary-encoded number.

The unsigned and signed Types

- Resolved array types of `std_ulogic`
  - Can be interpreted as `std_logic_vector` for arithmetic
  - Represent unsigned / two's complement binary integers

Resolved array types of std_ulogic
Can be interpreted as std_logic_vector for arithmetic
Represent unsigned / two's complement binary integers
Examples
signal a : unsigned(3 downto 0) := "1111"; -- 15

For illustration, consider the first example on the slide where the binary number 1111 is assigned to a signal of type `unsigned`. This results in the value being interpreted as decimal 15.

The unsigned and signed Types

- Resolved array types of `std_ulogic`
  - Can be interpreted as `std_logic_vector` for arithmetic
  - Represent unsigned / two's complement binary integers

- Examples

```
signal a : unsigned(3 downto 0) := "1111";  -- 15
```

Resolved array types of std_ulogic
- Can be interpreted as std_logic_vector for arithmetic
- Represent unsigned / two's complement binary integers

Examples

```
signal a : unsigned(3 downto 0) := "1111";  -- 15
signal b :   signed(3 downto 0) := "1111";  -- -1
```

The signed type on the other hand is interpreted as binary integer in two's complement encoding. Therefore, assigning the same binary number 1111 to the signal b, as shown on the slide, will result in an interpretation of the bit string as the value -1.

The unsigned and signed Types

- Resolved array types of std_ulogic
  - Can be interpreted as std_logic_vector for arithmetic
  - Represent unsigned / two's complement binary integers

- Examples

```
signal a : unsigned(3 downto 0) := "1111";  -- 15
signal b :   signed(3 downto 0) := "1111";  -- -1
```

Resolved array types of std_ulogic
Can be interpreted as std_logic_vector for arithmetic
Represent unsigned / two's complement binary integers
Bit string literal initialization possible
Examples
signal a : unsigned(3 downto 0) := "1111"; -- 15
signal b : signed(3 downto 0) := "1111"; -- -1

Note that as a consequence of the two types being merely arrays of `std_ulogic`, it is possible to assign them bit string literals.

The unsigned and signed Types

numeric_std
Motivation
IEEE Package
Types
Operators
Conversion

- Resolved array types of `std_ulogic`
  - Can be interpreted as `std_logic_vector` for arithmetic
  - Represent unsigned / two's complement binary integers
  - Bit string literal initialization possible

- Examples

```
signal a : unsigned(3 downto 0) := "1111";  -- 15
signal b :   signed(3 downto 0) := "1111";  -- -1
```

Resolved array types of std_ulogic
Can be interpreted as std_logic_vector for arithmetic
Represent unsigned / two's complement binary integers
Bit string literal initialization possible
Elements are nine-valued ⇒ useful for debugging
Examples
signal a : unsigned(3 downto 0) := "1111"; -- 15
signal b : signed(3 downto 0) := "1111"; -- -1

A further noteworthy consequence is that their elements can be any of the nine values of the IEEE 1164 standard. This is useful for debugging our designs.

The unsigned and signed Types

- Resolved array types of `std_ulogic`
  - Can be interpreted as `std_logic_vector` for arithmetic
  - Represent unsigned / two's complement binary integers
  - Bit string literal initialization possible
  - Elements are nine-valued ⇒ useful for debugging

- Examples

```
signal a : unsigned(3 downto 0) := "1111";   -- 15
signal b :   signed(3 downto 0) := "1111";   -- -1
```

Resolved array types of std_ulogic
Can be interpreted as std_logic_vector for arithmetic
Represent unsigned / two's complement binary integers
Bit string literal initialization possible
Elements are nine-valued ⇒ useful for debugging
Wrap around on over- / underflow ⇒ behavior of "real" hardware
Examples

However, probably the most important difference between the `integer` type and the numeric-standard types, is what we saw in the example of the previous slide. As you might know from other lectures, an adder circuit that overflows simply wraps around. This is what signals and variables of the `unsigned` and `signed` types do, thus allowing a more faithful description of arithmetic hardware.

The unsigned and signed Types

numeric_std
Motivation
IEEE Package
Types
Operators
Conversion

- Resolved array types of `std_ulogic`
  - Can be interpreted as `std_logic_vector` for arithmetic
  - Represent unsigned / two's complement binary integers
  - Bit string literal initialization possible
  - Elements are nine-valued ⇒ useful for debugging
- Wrap around on over- / underflow ⇒ behavior of "real" hardware
- Examples

```
signal a : unsigned(3 downto 0) := "1111";  -- 15
signal b :   signed(3 downto 0) := "1111";  -- -1
```

■ Most common operators are defined and implemented

Obviously, numerical data types are no good without operations defined on them. Therefore, the `numeric_std` package comes with a plethora of operations.

Arithmetic Operators

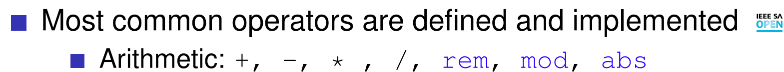
- Most common operators are defined and implemented

Most common operators are defined and implemented
Arithmetic: +, −, ∗ , /, rem, mod, abs

First and foremost, common arithmetic operations, listed on the slide, are defined for the `numeric_std` types. However, be aware that the division operator performs an integer division! To obtain the division remainder, the remainder operator, `rem`, can be used.

Arithmetic Operators

numeric_std
Motivation
IEEE Package
Types
Operators
Conversion



Most common operators are defined and implemented
Arithmetic: +, −, * , /, rem, mod, abs
Relational: >, <, <=, >=, =, /=
Logical: Same as for std_ulogic_vector
Shift / rotate: sll, srl, rol, ror, sla, sra

In addition to that, there are relational and logical operators defined, as well as the shift and rotate ones we already mentioned for the types of the `std_logic_1164` package. However, be aware that there is also the possibility to perform arithmetic shifts on the `unsigned` and `signed` types by using the `sla` and `sra` operators.

Arithmetic Operators

HWMod
WS25

numeric_std
Motivation
IEEE Package
Types
Operators
Conversion

- Most common operators are defined and implemented
  - Arithmetic: `+`, `−`, `*` , `/`, `rem`, `mod`, `abs`
  - Relational: `>`, `<`, `<=`, `>=`, `=`, `/=`
  - Logical: Same as for `std_ulogic_vector`
  - Shift / rotate: `sll`, `srl`, `rol`, `ror`, `sla`, `sra`

Most common operators are defined and implemented
Arithmetic: +, −, *, /, rem, mod, abs
Relational: >, <, <=, >=, =, /=
Logical: Same as for std_ulogic_vector
Shift / rotate: sll, srl, rol, ror, sla, sra
resize function

The package further provides a `resize` function that allows to create a longer or shorter value from a given one of type `unsigned` or `signed`. We will discuss this operator in more detail on the next slide.

Arithmetic Operators

numeric_std
Motivation
IEEE Package
Types
Operators
Conversion

- Most common operators are defined and implemented

  - Arithmetic: `+, −, * , /, rem, mod, abs`
  - Relational: `>, <, <=, >=, =, /=`
  - Logical: Same as for `std_ulogic_vector`
  - Shift / rotate: `sll, srl, rol, ror, sla, sra`
  - `resize` function

■ Most common operators are defined and implemented
■ Arithmetic: +, −, * , /, rem, mod, abs
■ Relational: >, <, <=, >=, =, /=
■ Logical: Same as for std_ulogic_vector
■ Shift / rotate: sll, srl, rol, ror, sla, sra
■ resize function
■ Useful overloads for integer / natural operand

Finally, in addition to implementations of the operators for the numerical types, there are useful overloads for applying the operators to a numerical type and an `integer` or `natural` operand. This allows to model operations with constants in a very concise manner.

Arithmetic Operators

- Most common operators are defined and implemented

  - Arithmetic: `+, −, * , /, rem, mod, abs`
  - Relational: `>, <, <=, >=, =, /=`
  - Logical: Same as for `std_ulogic_vector`
  - Shift / rotate: `sll, srl, rol, ror, sla, sra`
  - `resize` function
- Useful overloads for integer / natural operand

```
1 process is
2
3
4 begin
5
6
7
8
9
10
11
12
13
14
15
16   wait;
17 end process;
```

The `resize` function and the operator overloads deserve a bit more elaboration. We will discuss them by means of an example now, the skeleton of which is already shown on the slide.

Arithmetic Operators (cont'd)


```
1 process is
2
3
4 begin
5
6
7
8
9
10
11
12
13
14
15
16   wait;
17 end process;
```

```
1 process is
2   variable u : unsigned(3 downto 0) := "1010";
3   variable s :   signed(3 downto 0) := "1010";
4 begin
16  wait;
17 end process;
```


In the declarative section of the example process two four bit long variables of `unsigned` respectively `signed` type are declared. They are both initialized to the same value 1010. Let us now first discuss the `resize` function. As already mentioned, its purpose is to change the size of an argument of either `unsigned` or `signed` type. However, as you already know, array types in VHDL are value types. Thus, the length of an object of an array type cannot be changed after its declaration. Therefore, the function will return a new value of the desired length while keeping the original object unchanged. Furthermore, be aware that the `resize` function's behavior depends on the specific type of the argument that is to be resized. In the case of increasing the size of an `unsigned` value, the argument is extended with zeros to its left to match the desired length.

Arithmetic Operators (cont'd)

numeric_std
Motivation
IEEE Package
Types
Operators
Conversion

```
 1 process is
 2   variable u : unsigned(3 downto 0) := "1010";
 3   variable s :   signed(3 downto 0) := "1010";
 4 begin
 5
 6
 7
 8
 9
10
11
12
13
14
15
16   wait;
17 end process;
```

For example, consider the example on the slide which uses the function to get a version of `u` extended to 5 bits.

Arithmetic Operators (cont'd)


```
 1 process is
 2   variable u : unsigned(3 downto 0) := "1010";
 3   variable s :   signed(3 downto 0) := "1010";
 4 begin
 5   report to_string(resize(u, 5));
 6
 7
 8
 9
10
11
12
13
14
15
16   wait;
17 end process;
```

```
1 process is
2   variable u : unsigned(3 downto 0) := "1010";
3   variable s :   signed(3 downto 0) := "1010";
4 begin
5   report to_string(resize(u, 5));  -- 01010
6
7
8
9
10
11
12
13
14
15
16   wait;
17 end process;
```

Extending this with zero yields an `unsigned` object holding the bit string 01010. For a signed value on the other hand, the sign bit is used when extending.

Arithmetic Operators (cont'd)


```
1 process is
2   variable u : unsigned(3 downto 0) := "1010";
3   variable s :   signed(3 downto 0) := "1010";
4 begin
5   report to_string(resize(u, 5));  -- 01010
6
7
8
9
10
11
12
13
14
15
16   wait;
17 end process;
```

This is illustrated by this next example where the `resize` function is used to create a `signed` object that holds the value of the variable `s` extended by one bit. Since the sign bit is 1, the resulting value will also be extended by a 1.

Arithmetic Operators (cont'd)


```
1  process is
2    variable u : unsigned(3 downto 0) := "1010";
3    variable s :   signed(3 downto 0) := "1010";
4  begin
5    report to_string(resize(u, 5));  -- 01010
6    report to_string(resize(s, 5));  -- 11010
7
8
9
10
11
12
13
14
15
16   wait;
17 end process;
```

```
1 process is
2   variable u : unsigned(3 downto 0) := "1010";
3   variable s :   signed(3 downto 0) := "1010";
4 begin
5   report to_string(resize(u, 5));  -- 01010
6   report to_string(resize(s, 5));  -- 11010
7
8   report to_string(resize(u, 3));
9
10
11
12
13
14
15
16   wait;
17 end process;
```

In case the desired length passed to the `resize` function is shorter than the one of the array argument, the array must be truncated. This is the case for the current example shown on the slide. Again, the behavior differs depending on the specific type. For an `unsigned` value, bits are removed from left to right until the target length is reached.

Arithmetic Operators (cont'd)


```
1 process is
2   variable u : unsigned(3 downto 0) := "1010";
3   variable s :   signed(3 downto 0) := "1010";
4 begin
5   report to_string(resize(u, 5));  -- 01010
6   report to_string(resize(s, 5));  -- 11010
7
8   report to_string(resize(u, 3));
9
10
11
12
13
14
15
16   wait;
17 end process;
```

```
1  process is
2    variable u : unsigned(3 downto 0) := "1010";
3    variable s :   signed(3 downto 0) := "1010";
4  begin
5    report to_string(resize(u, 5));  -- 01010
6    report to_string(resize(s, 5));  -- 11010
7
8    report to_string(resize(u, 3));  -- 010
9
10
11
12
13
14
15
16   wait;
17 end process;
```


Thus, the result of the `resize`/ call shown on the slide is `u` with its left-bit dropped.

Arithmetic Operators (cont'd)


```
1  process is
2    variable u : unsigned(3 downto 0) := "1010";
3    variable s :   signed(3 downto 0) := "1010";
4  begin
5    report to_string(resize(u, 5));  -- 01010
6    report to_string(resize(s, 5));  -- 11010
7
8    report to_string(resize(u, 3));  -- 010
9
10
11
12
13
14
15
16   wait;
17 end process;
```

For signed values, bits are also dropped from left to right, but the most significant bit of the result will always be the sign bit of the original array. In contrast to the previous example with the `unsigned` truncation the result in the `signed` case is therefore 110, as this is the value of `s` with its left-most bit removed and the new left-most bit set to the old sign bit.

Arithmetic Operators (cont'd)

numeric_std
Motivation
IEEE Package
Types
Operators
Conversion

```
 1 process is
 2   variable u : unsigned(3 downto 0)  := "1010";
 3   variable s :   signed(3 downto 0)  := "1010";
 4 begin
 5   report to_string(resize(u, 5));   -- 01010
 6   report to_string(resize(s, 5));   -- 11010
 7
 8   report to_string(resize(u, 3));   -- 010
 9   report to_string(resize(s, 3));   -- 110
10
11
12
13
14
15
16   wait;
17 end process;
```

Finally, let us briefly illustrate the useful operator overloads we mentioned before. As shown in the example code on the slide, we can simply perform operations between variables of `unsigned` respectively `signed` type and `integer` values. This allows to increment `u` by 1 and subtract 2 from `s` in a very concise manner. Furthermore, the example also shows uses of the respective overloads for two of the relational operators. Again, this allows for very concise comparisons between the numerical array types and integer constants.

Arithmetic Operators (cont'd)


```
1  process is
2    variable u : unsigned(3 downto 0) := "1010";
3    variable s :   signed(3 downto 0) := "1010";
4  begin
5    report to_string(resize(u, 5));  -- 01010
6    report to_string(resize(s, 5));  -- 11010
7
8    report to_string(resize(u, 3));  -- 010
9    report to_string(resize(s, 3));  -- 110
10
11   u := u + 1;
12   s := s - 2;
13   if u = 0 or s < -1 then
14     report "HWMod is awesome";
15   end if;
16   wait;
17 end process;
```

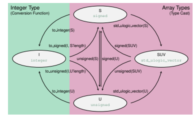
Conversion function from / to integer, type casts between array types

Finally, let us briefly discuss the conversion between the different types we know by now. In principle, we have to distinguish between the array types of the numeric_std and std_logic_1164 packages and basic integers. Between the different arrays, std_ulogic_vector, std_logic_vector, signed and unsigned we can simply perform type casts, as the internal representation is the same, and we just want to change its interpretation. If we want to convert from the numerical types of the numeric_std package to the integer type, or vice versa, we require conversion functions.

Type Conversion

numeric_std
Motivation
IEEE Package
Types
Operators
Conversion

Conversion function from / to integer, type casts between array types

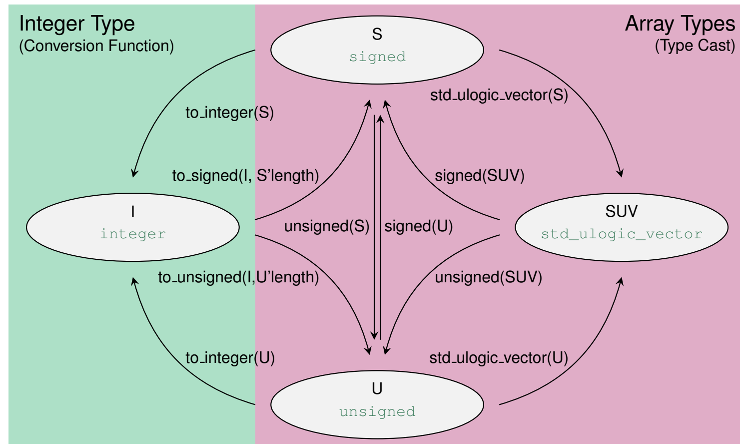


Have a look at the image on the slide which shows all possible type conversions between these types. Note how all conversions in the red-shaded area are just type casts between the different array types, and how the green-shaded area marks contains conversions functions. Note how this essentially partitions the conversions into those from `integer` to the `numeric_std` types and vice versa, and those between the different array types of `std_ulogic`. This difference is also reflected by the naming convention. Where type-casts simply use the target type name, the conversion functions start with a `to`. Furthermore, note how the conversion from an `integer` to a `numeric_std` type requires a length. This makes sense considering that the `numeric_std` types are arrays type that require a range constraint and that an `integer` might also hold values smaller than its full 32-bit would support.

Type Conversion

numeric_std
Motivation
IEEE Package
Types
Operators
Conversion

Conversion function from / to `integer`, type casts between array types



Lecture Complete!

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

numeric_std
Motivation
IEEE Package
Types
Operators
Conversion

Lecture Complete!