

## Data Types

```
library IEEE;
use IEEE.std_logic_1164.all;

entity FULLADD is
    port (A, B, CIN : in bit;
          SUM, CARRY : out bit);
end FULLADD;

architecture STRUCT of FULLADD is
    signal I1, I2, I3 : bit;
    component HALFADD
        port(A,B : in bit;
             SUM, CARRY : out bit);
    end component;

    component ORGATE
        port(A,B : in bit;
             Z : out bit);
    end component;

begin
    u1:HALFADD port map(A,B,I1,I2);
    u2:HALFADD port map(I1,CIN,SUM,I3);
    u3:ORGATE port map(I3,I2,CARRY);
end STRUCT;

configuration CFG_STR of FULLADD is
    for STRUCT
    end for;
end CFG_STR;
```

VHDL is referred to as a strong – typed language. Each object in VHDL (signal, variable or constant) must have a data type defined during an object declaration. Such a data type presents the set of values for a signal, variable or constant.

```
SUM <= CIN xor A xor B;
```

In each assignment, the types at the both side of assignment operator should be the same.

## Standard Data Types

Type bit is ( '0', '1' ).

The quotes are essential, because the values are characters. The operators that apply to type bit are:

- Logical: not, and, or, nand, nor, xor, *xnor*
- Comparison =, /=, <, <=, >, >=

For logical operators the result has the type bit. Type bit is represented by signal wire, with the value '0' represented by logic 0 and the value '1' by logic '1'.

```
CARRY <= (A and B) or (CIN and A) or (CIN and B).
```

Operator *not* has a higher precedence, operators *not*, *and*, *or*, *nand*, *nor*, *xor*, *xnor* have the same precedence, so in the expressions:

```
z <= (a and b) or (c and d);
y <= a or (b and c);
```

the brackets are necessary.

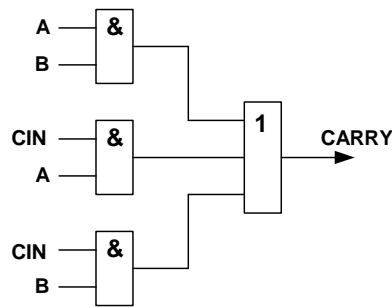


Figure 1. Intermediate signals in bit expressions.

In practice, type bit is very rarely used. Typically at least four logic levels are required ('0', '1', 'Z' – *high impedance* and 'X' – *unknown*).

The nine value type *std – logic* is usually used instead of bit.

Type Boolean is (true, false)

The operators that apply to type Boolean are:

- Logical: not, and, or, nand, nor, xor, *xnor*
- Comparison =, /=, <, <=, >, >=

Type Boolean is usually used for comparison between two values of any other types.

**a = b** The result is a Boolean value for any data type of a, b.

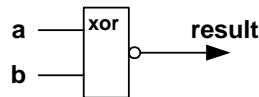


Figure 2. Logic mapping of Boolean equality.

The result with Boolean type as a rule is used in such a constructions:

```

if a = b then
    result <= '1';
else
    result <= '0';
end if;

```

### Integer type

Built in numeric type. This type is range -2147483648 to +2147483647 ( $-2^{31}$  to  $2^{31} - 1$ )

Operators:

- Comparison =, /=, <, <=, >, >=
- Arithmetic sign +, sign -, abs, +, -, \*, /, mod, rem, \*\*

### User – Defined Integers

type short is range -128 to 127;

In integer expression it is not possible to mix different integer types (strong types). It is not a good practice to define a lot of unique types for each signal in a design. If we use type short, defined above, the result of integer calculation must be within the range of the type (-128 to 127). Otherwise – the error will occur during simulation. When an integer type is defined, the following operators can be applied to the new type:

comparison =, /=, <, <=, >, >=

arithmetic sign +, sign -, abs, +, -, \*, /, mod, rem, \*\*

### Integer Subtypes

subtype natural is integer range 0 to integer'high;

subtype positive is integer range 1 to integer'high;

integer'high =  $2^{31} - 1$

When, for example, type natural is used in calculation, the calculation are carried out using the base type, integer, and then checked to ensure that they fit to natural. This check is not carried out until an assignment is made (even for subtypes).

#### Example:

Subtype nat4 is natural range 0 to 15;

w, x, y, z: nat4;

w <= x - y + z; -- x = 3; y = 4; z = 5

3 - 4 = -1, but at the end w = -1 + 5 = 4 (nat4).

### Character type

type character is (-- ascii set);

### Time type

*time* is a special data type as it consists out of a numerical value and a physical unit. It is used to delay the execution of statements for a certain amount of time, e.g. in test benches or to model gate and propagation delays. Signals of data type *time* can be multiplied or divided by *integer* and *real* values. The result of these operations remains of data type *time*.

```
signal clk: bit;
constant clk_period: time := 10 ns;
...
wait for clk_period;
...
wait for clk_period * 4;
...
clk <= not clk after clk_period;
```

Available time units: fs, ps, ns, us, ms, sec, min, hr.

## Enumeration type

Enumeration type is a type composed of a set of names.

Example:

```
type opcode is (add, sub, mult, div, shl, shr);
signal instruction : opcode;
```

Synthesis tools encode such values *add*, ..., *shr* by binary vectors with a minimal number of components – three in this example:

```
add = "000"    div = "011"
sub = "001"    shl = "100"
mult = "010"   shr = "101"
```

At the beginning of simulation signal instruction is equal to *add* – to the leftmost value in the type definition. If we would like to use other encoding we should change the object order in the type definition. For example, for such an encoding:

```
add = "001"    div = "100"
sub = "011"    shl = "111"
mult = "000"   shr = "110"
```

the corresponding type definition is:

```
type opcode is (mult, add, empty1, sub, div, empty2, shr, shl).
```

We will not use empty1 and empty2 as instructions in our design. It is forbidden to use the same name in the enumeration type.

Only comparison operators are predefined for an enumeration type:

comparison    =, /=, <, <=, >, >=

The comparison operators are defined in terms of the position values. The first (or left) literal in the type is regarded as the smallest value and the last (or right) as the largest one.

Types *Bit* ('0', '1') and *Boolean* (*true*, *false*) are the two standard enumeration types.

## Multi - values logic type - std\_logic

Type std\_logic is

```
(
  'U', -- Uninitialized
  'X', -- Forcing Unknown
  '0', -- Forcing 0
  '1', -- Forcing 1
  'Z', -- High Impedance
  'W', -- Weak Unknown
  'L', -- Weak 0
  'H', -- Weak 1
  '-', -- Don't care
);
```

std\_logic is not part of VHDL, but it is an IEEE standard extension to the language under standard number 1164. It exists in a library called IEEE in a package called std\_logic\_1164.all.

Place this before entity or architecture where this type is used:

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

For synthesis only three values {'0', '1', 'Z'} are used. 'Z' – high impedance value which is used in tristates. Some synthesis tools used '-' (don't care) for optimization.

### Arrays

Array is a collection of multiple elements with the same type.

Syntax: type *type\_name* is array (range) of *element type*;

```
type nibble is array (3 downto 0) of std_logic;  
  
type mem is array (0 to 7) of nibble;  
  
signal a_bus : nibble;  
  
signal mem0 : mem;
```

An array type should be declared before declaration of an object.

```
type three_val is ('0', '1', 'x');  
  
type my_vector is array (natural range <>) of three_val;  
  
signal my_byte : my_vector (7 downto 0);
```

Unconstrained array type allow to declare different-size objects and use these objects through each other.

Two predefined arrays are in VHDL:

- bit\_vector – array of bits;
- string – array of characters.

```
signal bus1: bit_vector (3 downto 0);  
constant message1 : string := "Test 1 Completed";
```

```
signal a: std_logic_vector (3 downto 0);  
signal b: std_logic_vector (1 to 4);  
signal c: std_logic_vector (0 downto 3); NOT CORRECT  
signal d: std_logic_vector (1 downto 4); NOT CORRECT
```

A signal assignment with array is made element by element from left to right.

```
signal up: std_logic_vector (1 to 4);  
signal down: std_logic_vector (4 downto 1);  
    ...  
  
    -- up <= down means:  
  
    up(1) <= down(4);  
    up(2) <= down(3);  
    up(3) <= down(2);  
    up(4) <= down(1);
```

In the assignment, two arrays should have the same size (but maybe the different ranges).

Use of slices:

```
up(2 to 3) <= down(4 downto 3);
```

### Concatenations and Aggregates

```
signal z_bus, a_bus, b_bus :  
    bit_vector(3 downto 0);  
signal a, b, c, d : bit;  
signal byte : bit_vector(7 downto 0);  
  
z_bus <= a & b & c & d;  
byte <= a_bus & b_bus;
```

Concatenation can be used only at the right side of assignment. The concatenation operator (&) allows to construct a long array from smaller arrays and elements.

```
signal z_bus : bit_vector(3 downto 0);  
signal a, b, c, d : bit;
```

```
z_bus <= (a, b, c, d);
```

It is equivalent to

```
z_bus(3) <= a;  
z_bus(2) <= b;  
z_bus(1) <= c;  
z_bus(0) <= d;
```

One more method of assignment to the elements of array is an *aggregate*. Aggregate is written in the parenthesis. Assignments to each element are separated by comma.

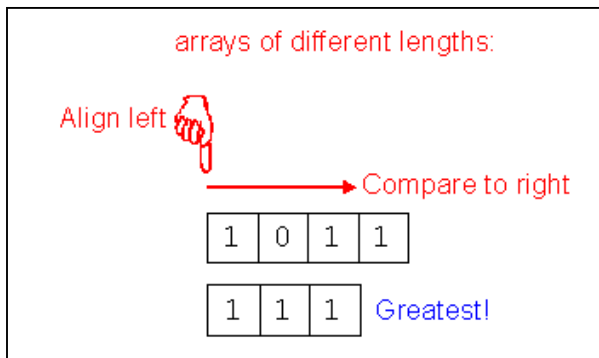
```
signal x : bit_vector(3 downto 0);  
signal a, b, c, d : bit;  
  
x <= (3 => '1', 1 downto 0 => '0', 2 => b);  
  
x <= (3 => '1', 2 => b, others => '0');  
  
-- reset of x:  
  
x <= (others => '0');
```

Assignment by name. The same value can be assign in the range.

1. a <= "1000"
2. a <= (3 => '1', 2 => '0', 1 => '0', 0 => '0');
3. a <= (3 => '1', 2|1|0 => '0');
4. a <= (3 => '1', others => '0');
5. a <= (3 => '1', 2 downto 0 => '0');
6. a <= ('1', '0', '0', '0');
7. a <= ("1000");

7 methods for the same assignment

**comparison** =, /=, <, <=, >, >=



If such a comparison is undesired then we can use right alignment for arrays with the same length:

'0' & "111" < "1011"

**Logical operators:** not, and, or, nand, nor, xor, xnor

```
signal a, b, z : std_logic_vector (3 downto 0);
```

-- Assignment:

```
z <= a and b;
```

-- means:

```
z(3) <= a(3) and b(3);  
z(2) <= a(2) and b(2);  
z(1) <= a(1) and b(1);  
z(0) <= a(0) and b(0);
```

Logical operations can be apply to arrays with the same type and with the same length. The operator is matching elements by position returning an array of the same length.