Digital Design with VHDL

Type Overview

By Benjamin Abramov

All Rights reserved ©
No duplication copying and distribution of this document is allowed without the proper authorization of the author

Typical VHDL file structure

Library's and packages declaration

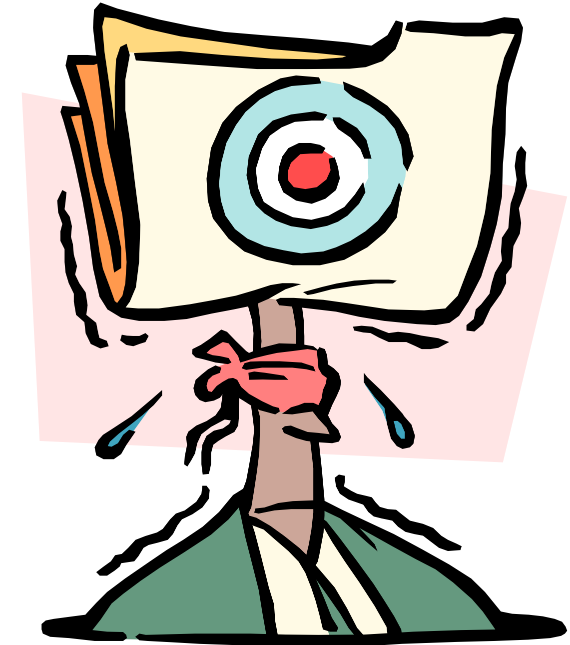
Entity declaration

Architecture

Configuration

© All rights reserved to Abramov B.

Topic Two



- Data Assignment 69
- Scalar types.........................79
- String 87
- User defined types.................88
- Resolved types.....................93
- Subtypes...........................112
- Type conversions.................116
- Expressions and Operators.......121

© All rights reserved to Abramov B.

Data Assignment

For assignment operation VHDL uses the combination of symbols – **<=**

- An assignment to a port defines a driver to that port.
- A standard port has only one source (driver).
- **Assignment can not be done between different types!**

© All rights reserved to Abramov B.

Data Assignment

Examples:

```
single_bit, one_bit          : bit;
integer_val                  : integer range 15 downto 0;
boolean_Val                  : boolean;
multiple_bit, bus_vector : bit_vector(15 downto 0);
--------------------------------------------------------------------------------
single_bit                   <= '1';
one_bit                      <= single_bit; -- one_bit = '1'
integer_val                  <= 12;
integer_val                  <= 30;     --wrong, to large value, over the range
boolean_val                  <= true;
multiple_bit                 <= "0000000000000000";
  -- assigning multiple_bit the value "1111111100000000"
multiple_bit(15 downto 8) <= x"ff";
  -- assigning multiple_bit the value "1111111100000001";
multiple_bit(0)              <= single_bit;
bus_vector                   <= multiple_bit;
```

© All rights reserved to Abramov B.

Data Assignment

For bus assignment VHDL supports two different forms of assignment: ***aggregate and concatenation***.

Aggregate is a grouping of values in order to form an array.

```
signal a, b, c, d : bit;
nibble : bit_vector(3 downto 0);
a <= '1';
b <= '0';
c <= '1';
d <= '0';

nibble <= (a, b, c, d);     -- nibble = "1010";
```

Equivalent assignment is :

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

© All rights reserved to Abramov B.

Data Assignment

Aggregate

Positional Association

Positional association – values are associated with elements from left to right.

nibble<=(‘0’,’1’,’0’,’1’); -- nibble=”0101”

Named Association

Named association – values are explicitly referenced and order is not important.

nibble <= (2 => a, 1 => b, 0 => d, 3 => c); -- nibble=”1100”

© All rights reserved to Abramov B.

Data Assignment

With positional association , elements may be grouped together using the | (pipe) symbol or a range.

The keywords **others** may be used to refer to all elements.

```
nibble <= (3 | 2 => ’1’, others => ‘0’);        -- nibble = ”1100”;
nibble <= (3 downto 1 => ’0’, 0 => ’1’);   -- nibble = ”0001”;
nibble <= (others => ’0’);                 -- nibble = ”0000”;
```

© All rights reserved to Abramov B.

Data Assignment

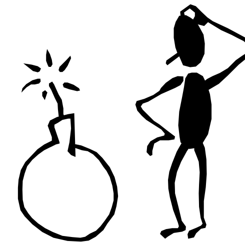
Another kind of aggregate is grouped assignment, i.e. assigning value to a number of elements of the same type:

signal a, b, c, d : **bit**;

(a, b, c, d) <= **bit_vector’**(x”5”);

Equivalent assignment is :

a <= ‘0’;
b <= ‘1’;
c <= ‘0’;
d <= ‘1’;



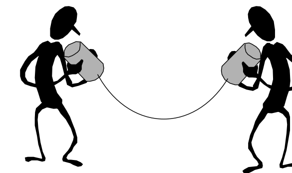
© All rights reserved to Abramov B.

Data Assignment

Concatenation

The concatenation operator (denoted as &) composes two one-dimensional arrays into one larger array of the **same type**.

bus_vector : **bit_vector**(7 **downto** 0);



nibble <= a & b & c & d; -- nibble="1010";
bus_vector <= nibble(2 **downto** 1) & nibble(0) & nibble(3) & "0010";

© All rights reserved to Abramov B.

Data Assignment

Constant values -

```
a_bus     : bit_vector(7 downto 0);
new_bus : bit_vector(5 downto 0);

a_bus     <=   “01011100”;   -- binary value by default
new_bus <=  b”010111”;        -- explicit binary value
a_bus     <=  b“0111_0101”; -- binary value with nibble separator
new_bus <=  o”67”;              -- octal base of value presentation
a_bus     <=  x”a5”;              -- hexadecimal  base
```

- Be Careful !!!

```
b”0”    -- 1 bit with value 0
o”0”    -- 3 bits with value 0, same as B”000”
x”0”    -- 4 bits with value 0, same as B”0000”
```

b”0” ≠ o”0” ≠ x”0”

© All rights reserved to Abramov B.

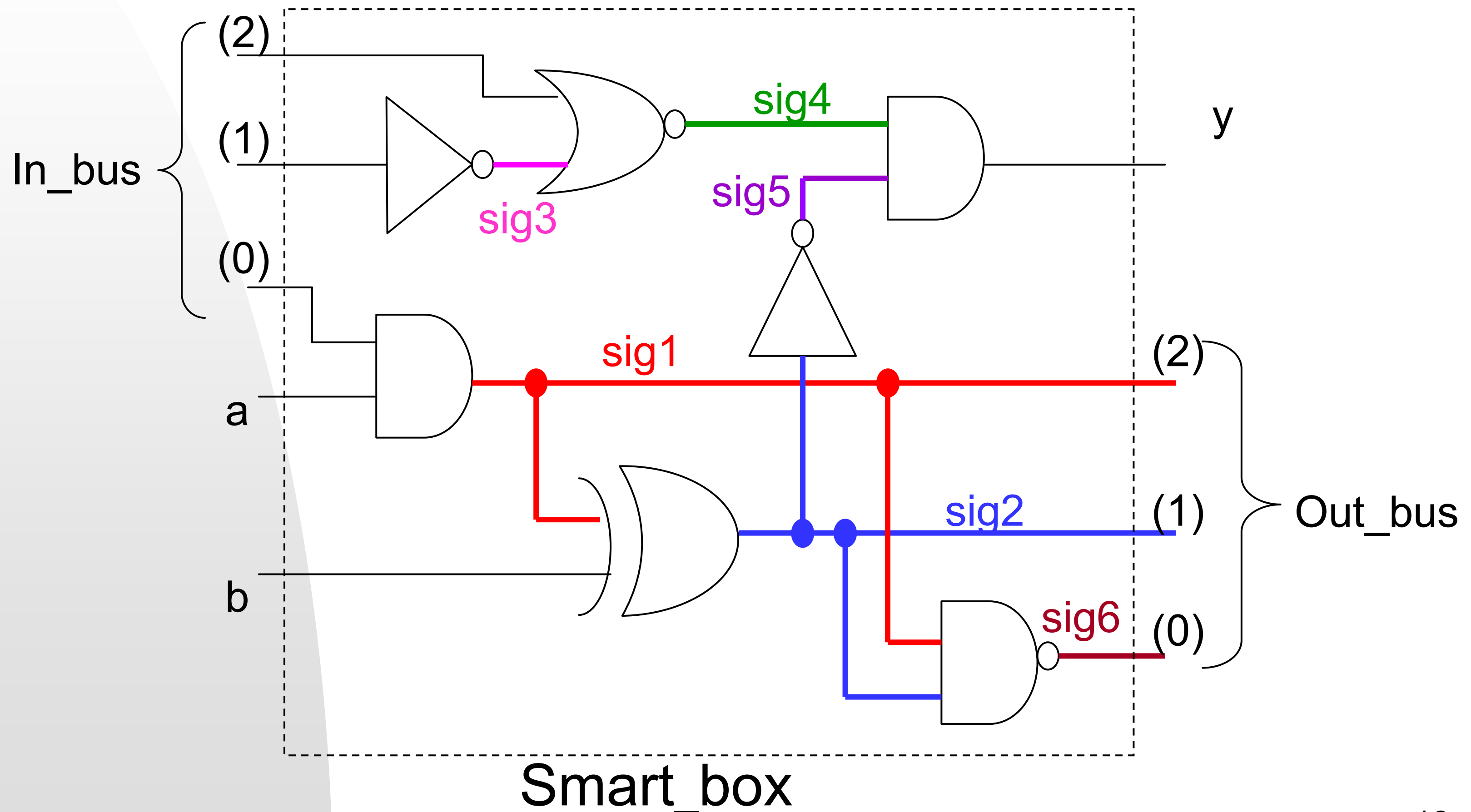
Data Assignment

- Integer assignment 764563254 -- default is decimal
- Other bases: base#number# base $\in \{2,3,..,16\}$
  - 16#F3A4# -- hex $(F3A4)_{16}$
  - -2#01101101# -- binary $-(01101101)_2$
  - 3#20120112# -- ternary $(20120112)_3$
- Scientific Notation: number E± exponent
  - 1372E+11 -- 1372 x (10**11)
  - 16#F3A4#e+9 -- hex $(F3A4)_{16}$ x (10**9)
- Improve readability using '_'
  - 2#0110_1101# -- binary $(01101101)_2$
  - 16#F3_A4#e+9 -- hex $(F3A4)_{16}$ x (10**9)
- Real Numbers: integers + '.'
  - -23432.123 -- decimal -23432.123
  - 2#01.10_1101# -- binary $(01.101101)_2$
  - 16#0.F3_A4#e+9 -- hex $(0.F3A4)_{16}$ x (10**9)

© All rights reserved to Abramov B.

Self Work

Write entity and architecture for the following design using signals.
Use a. concatenation
b. aggregate
For constructing out_bus port.



© All rights reserved to Abramov B.

Scalar types

Scalar types are ordered types that can hold only one value at a time.

Scalar types include four categories :

- Enumerated types such as boolean and bit.
- Integer types – whole numbers with integer range of :
  (–(2**31) + 1) to ((2**31 – 1).
- Floating Types – approximate representation of real numbers.
  The precision of the approximation is not defined by the VHDL language standard, however it must be at least six decimal digits.
  Floating type range is : -1E38 to +1E38
- Physical types – representing measurement units.

© All rights reserved to Abramov B.

Scalar types

Creation of a floating point type is declared using the syntax:

floating_type_definition := range_constraint

Some examples are:

type signal_level **is range** -10.00 **to** +10.00;
type probability **is range** 0.0 **to** 1.0;

The real type is predefined by the language.
The range of this type is implementation defined, though it is guaranteed that its value is within -1E38 to +1E38 range.

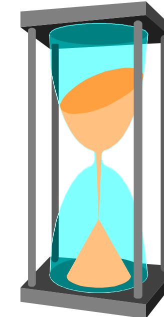
Note: Real type is not supported by synthesis tools.

© All rights reserved to Abramov B.

Scalar types

The predefined physical type **time** is very important in VHDL, as it used extensively to specify delay and time constraints in simulation.

```
type time is range –(2**31) to (2**31)
    units
        fs;
        ps   =  1000 fs;
        ns   =  1000 ps;
        us   =  1000 ns;
        ms   =  1000 us;
        sec  =  1000 ms;
        min  =   60   sec;
        hr   =   60   min;
    end units;
```



© All rights reserved to Abramov B.

Scalar types

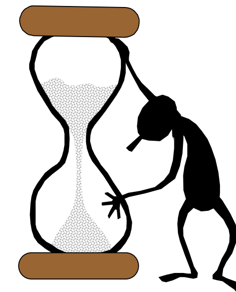
VHDL provides a predefined function ***now*** that returns the current simulation time.

The **pure function** now **returns** delay_length;

delay_length is predefined subtype of the physical type **time**, constrained to non-negative **time** value.

The time type is not supported by synthesis tools.

It is used in test_benches and modeling.



© All rights reserved to Abramov B.

Scalar types

The ***character*** data type enumerates the ASCII character set.

Nonprinting characters are represented by a three-letter name, such as NUL for the null character.

Printable characters are represented by themselves, in single quotation marks, as follows:

CHARACTER_VAR: **character**;

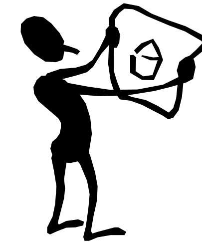
. . .

CHARACTER_VAR <= 'A';

© All rights reserved to Abramov B.

Scalar types

```
type character is (
    NUL,  SOH,  STX,  ETX,  EOT,  ENQ,  ACK,  BEL,
    BS,   HT,   LF,   VT,   FF,   CR,   SO,   SI,
    DLE,  DC1,  DC2,  DC3,  DC4,  NAK,  SYN,  ETB,
    CAN,  EM,   SUB,  ESC,  FSP,  GSP,  RSP,  USP,
    ' ',  '!',  '"',  '#',  '$',  '%',  '&',  ''',
    '(',  ')',  '*',  '+',  ',',  '-',  '.',  '/',
    '0',  '1',  '2',  '3',  '4',  '5',  '6',  '7',
    '8',  '9',  ':',  ';',  '<',  '=',  '>',  '?',
    '@',  'A',  'B',  'C',  'D',  'E',  'F',  'G',
    'H',  'I',  'J',  'K',  'L',  'M',  'N',  'O',
    'P',  'Q',  'R',  'S',  'T',  'U',  'V',  'W',
    'X',  'Y',  'Z',  '[',  '\',  ']',  '^',  '_',
    '`',  'a',  'b',  'c',  'd',  'e',  'f',  'g',
    'h',  'i',  'j',  'k',  'l',  'm',  'n',  'o',
    'p',  'q',  'r',  's',  't',  'u',  'v',  'w',
    'x',  'y',  'z',  '{',  '|',  '}',  '~',  DEL);
```



© All rights reserved to Abramov B.

Scalar types

Type **character** is an example of an enumeration type containing a mixture of identifiers and characters.

The ***string*** data type is an unconstrained array of characters.

string value is enclosed in double quotation marks as follows:

```
header: string(1 to 10);
header <= ”start_time”;
```

Also, the characters '0' and '1' are members of both **bit** and **character** types. When '0' or '1' occur in a program, the context will be used to determine which type is being used.

© All rights reserved to Abramov B.

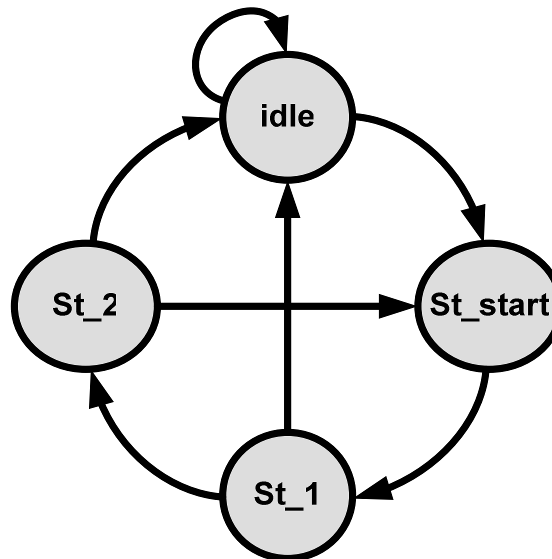
User defined types

A useful type in VHDL is the user defined **enumeration** type.

Enumeration - literal values ordered and numbered in a list.

type sm_type **is** (idle, st_start, st_1, st_2);

The Enumerated type is very useful for encoding state machine description



© All rights reserved to Abramov B.

User defined types

The hardware implementation of the above types will result with a **bus** creation. The Width of the bus created is depended upon the ***encoding style.***

For example :

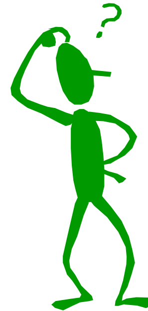
type sm_type **is** (idle, st_start, st_1, st_2);

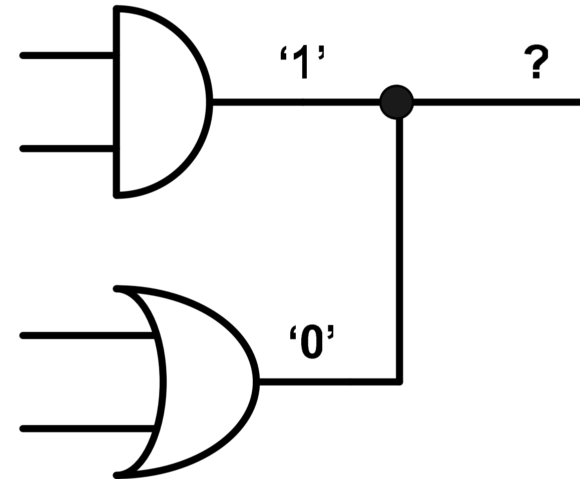
State name	Encoding style		
	Binary	Gray	One_Hot
idle	"00"	"00"	"0001"
St_start	"01"	"01"	"0010"
St_1	"10"	"11"	"0100"
St_2	"11"	"10"	"1000"

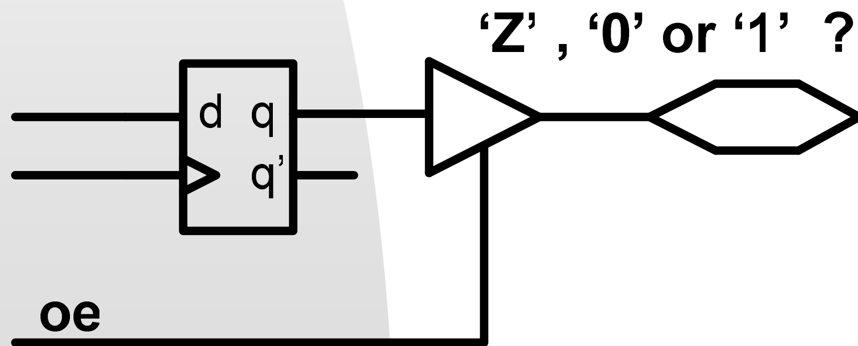
© All rights reserved to Abramov B.

Resolved type

What is happening at these circuits ?



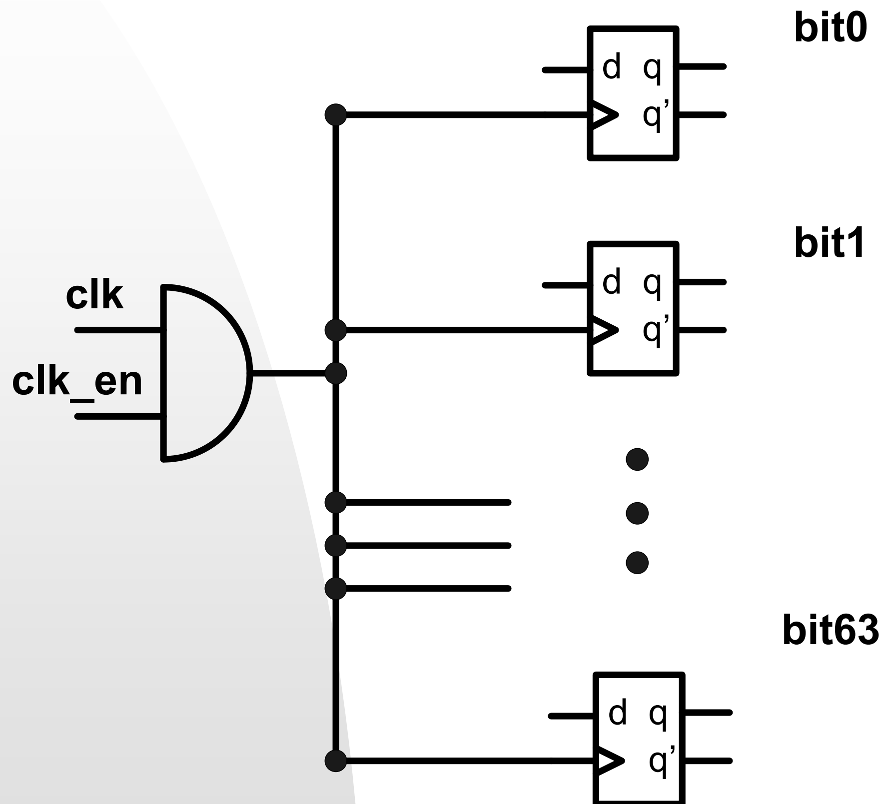


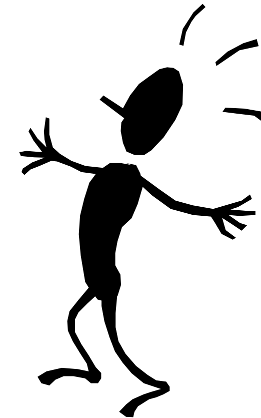


© All rights reserved to Abramov B.

Resolved type

What will happen with the clock signal when it will get ***high fan-out*** ?

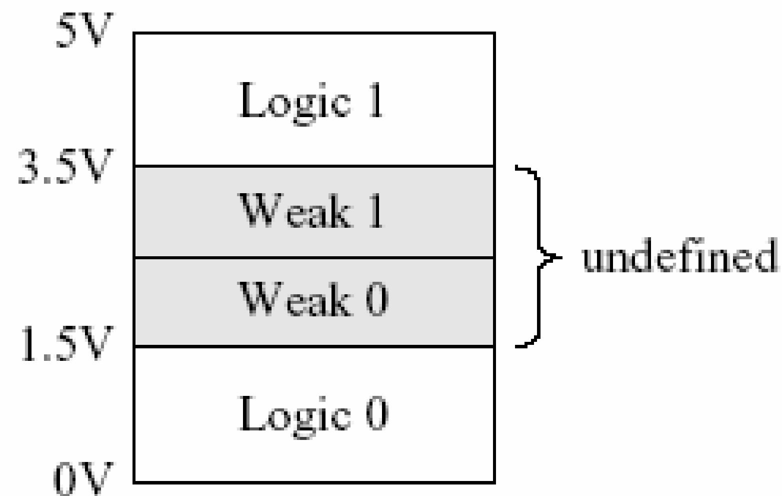




© All rights reserved to Abramov B.

Resolved type

Voltage levels (TTL logic) for logic 1 and 0.



What occurs when the entrance signal accepts undefined values ?

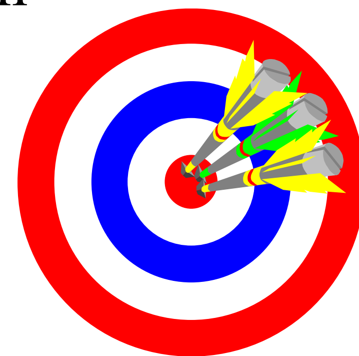
© All rights reserved to Abramov B.

Resolved type

Type bit is mainly used for abstract models where no concern about the electrical behavior of the signals is necessary.

The IEEE standard package ***std_logic_1164*** defined the enumerated type: **std_ulogic** which is a more realistic behavioral description to a signal then the bit type.

The package also defines the **std_ulogic_vector** type which describes a one-dimensional array type with each element being of type **std_ulogic**



The main type of really hardware model.

© All rights reserved to Abramov B.

Resolved type

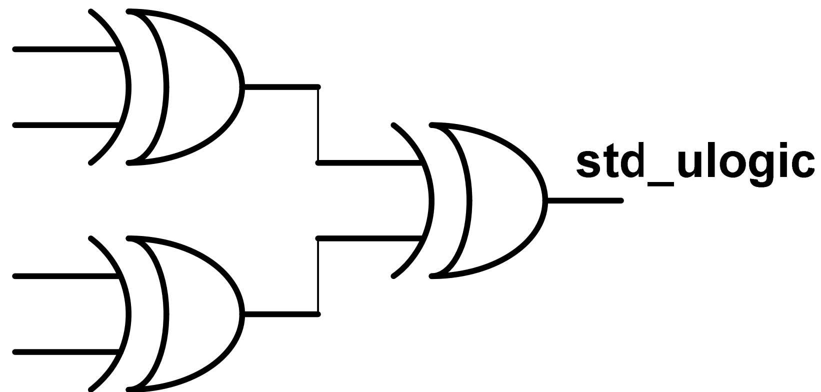
```
type std_ulogic is
        ( 'U', -- uninitiated
          'X', -- forcing unknown
          '0', -- forcing zero
          '1', -- forcing one
          'Z', -- high impedance
          'W', -- weak unknown
          'L', -- weak zero
          'H', -- weak one
          '-'  -- don't care
        );
```

This type of logic is called ***MVL – Multi Valued Logic***.

© All rights reserved to Abramov B.

Resolved type

Signal of type std_ulogic is defined to have only one driver!
(like the following XOR gate for example)



© All rights reserved to Abramov B.

Resolved type

Signals having more then one driver must be of a resolved type!!!

The basic resolved type added to VHDL is **std_logic** type which is a resolved **std_ulogic**. The defined type also handles the resolved vector type **std_logic_vector**. **Std_logic** and **std_logic_vector** are also defined in the std_logic_1164 package mentioned earlier and handles the std_ulogic type.

In order to use std_logic and std_ulogic types it is necessary to declare a library access and the relevant package.

```
library IEEE;
use IEEE.std_logic_1164.all;
---------------------------------
entity any_entity is
        port(
                a : out std_logic;
                b : in  std_logic;
                c : in  std_logic
             );
end entity any_entity;
```

© All rights reserved to Abramov B.

Resolved type

A *resolved* type is a type with a *resolution function!!!*

The resolution function of type **std_logic** takes a vector of the (unresolved) base-type of std_logic : **std_ulogic**.

It returns a single of type **std_logic**.

Now if we have two concurrent assignments to any signal of type **std_logic**, the resolution function will be called to determine the final value of the signal.

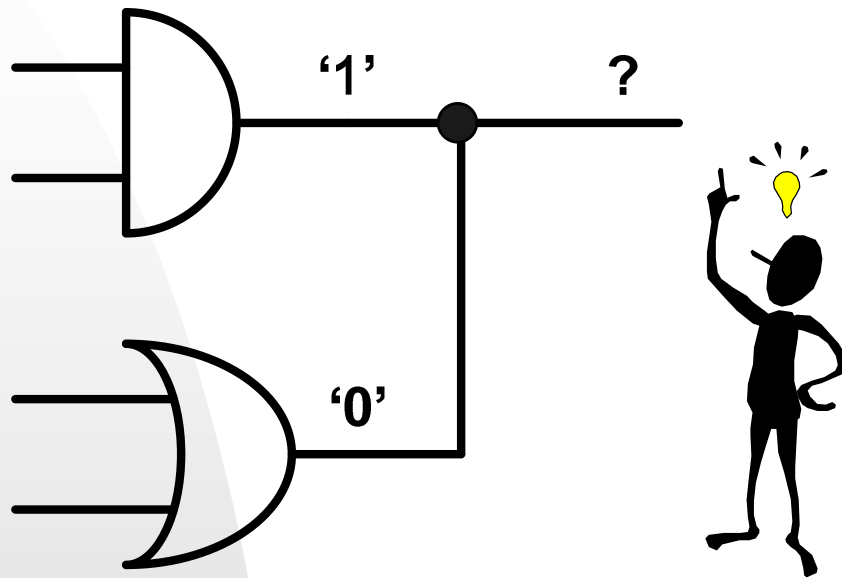
© All rights reserved to Abramov B.

Resolved type

U	**X**	**0**	**1**	**Z**	**W**	**L**	**H**	**-**	
U	U	U	U	U	U	U	U	U	**U**
U	X	X	X	X	X	X	X	X	**X**
U	X	0	X	0	0	0	0	X	**0**
U	X	X	1	1	1	1	1	X	**1**
U	X	0	1	Z	W	L	H	X	**Z**
U	X	0	1	W	W	W	W	X	**W**
U	X	0	1	L	W	L	W	X	**L**
U	X	0	1	H	W	W	H	X	**H**
U	X	X	X	X	X	X	X	X	**-**

© All rights reserved to Abramov B.

Resolved type



The result of this description is = 'X'.

© All rights reserved to Abramov B.

Resolved type

STD_LOGIC and **STD_ULOGIC** ***can*** be assigned to each other:

```
signal sl   : std_logic;
signal sul : std_ulogic;

sl   <= sul; --correct;
sul <= sl;   --correct;
```

STD_LOGIC_VECTOR and STD_ULOGIC_VECTOR
can not **be assigned to each other:**

```
signal slv   : std_logic_vector(7 downto 0);
signal sulv : std_ulogic_vector(7 downto 0);

slv   <= sulv; -- wrong;
sulv <= slv;   -- wrong;
```

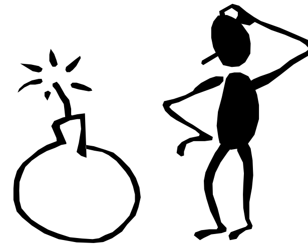
© All rights reserved to Abramov B.

Resolved type

Another useful package is the ***STD_LOGIC_ARITH*** package.
The package ***STD_LOGIC_ARITH*** defines two new data types, both are arrays of **STD_LOGIC.**

UNSIGNED => Positive only

SIGNED => Positive and Negative (In two's compliment format)



© All rights reserved to Abramov B.

Resolved type

Package ***STD_LOGIC_ARITH*** contains arithmetic and logical operators for combinations of vectors and integers, along with useful functions for converting between them using overloading.

The ***STD_LOGIC_ARITH*** package ***does not*** contain arithmetic operation for **std_logic_vectors** and **std_ulogic_vectors** !

Only for signed and unsigned types.

© All rights reserved to Abramov B.

Resolved type

```
signal A : std_logic_vector(3 downto 0);
signal B : unsigned(3 downto 0);


    b <= a + 7; -- wrong

    b <= unsigned(a) + 7; -- correct
```

Casting a into an unsigned vector enables the requested addition operation.

© All rights reserved to Abramov B.

Resolved type

Packages ***STD_LOGIC_SIGNED*** and ***STD_LOGIC_UNSIGNED*** include a set of signed/unsigned arithmetic and conversion functions for **std_logic_vectors**.

These 2 packages are automatically performing casting into unsigned or signed types.

Only one package STD_LOGIC_SIGNED or STD_LOGIC_UNSIGNED may be used for a specific design!

- Designs that have only positive vectors will use the **std_logic_unsigned** package.
- Designs that have positive and negative vectors will use the **std_logic_signed** package.

© All rights reserved to Abramov B.

Resolved type

Example :

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity comp is
	port (a,b	: in  std_logic_vector(7 downto 0);
		comp_out : out boolean);
end entity comp;
architecture arc_comp of comp is
begin
   comp_out <= ( a > b );
end architecture arc_comp
```

when:
a = x”07”
b = x”82”
comp_out is **false**

© All rights reserved to Abramov B.

Resolved type

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
entity comp is
      port (a,b          : in  std_logic_vector(7 downto 0);
               comp_out : out boolean);
end entity comp;
architecture arc_comp of comp is
begin
    comp_out <= ( a > b );
end architecture arc_comp;
```

when:

a = x”07”

b = x”82”

comp_out is **true**

© All rights reserved to Abramov B.

Subtypes

Subtype – defines a new type that contains parts of the values of a previously defined type.

subtype subtype_name **is** base_type **range** range_constraint**;**

Example (user-defined subtypes):

subtype nibble **is** **std_logic_vector(**3 **downto** 0**);**

subtype byte **is** **std_logic_vector(**7 **downto** 0**);**

© All rights reserved to Abramov B.

Subtypes

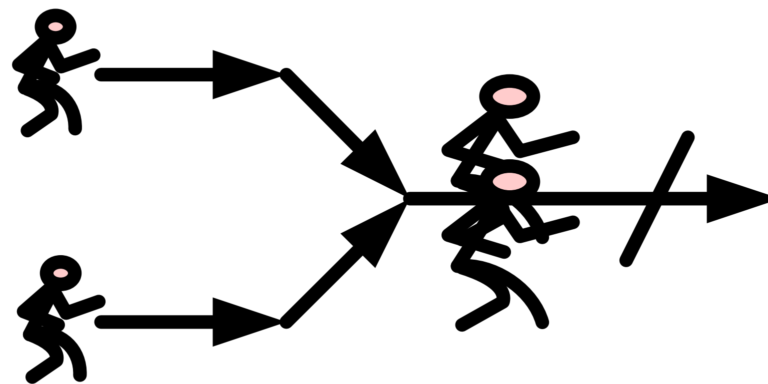
A subtype object can be assigned to its base type object:

```
subtype byte is std_logic_vector(7 downto 0);
subtype short is integer range (2**16 –1) downto 0;
signal my_byte     : byte;
signal new_byte    : std_logic_vector(7 downto 0);
signal new_word    : std_logic_vector(15 downto 0);
signal my_natural : natural;
signal my_int      : integer;
my_byte        <= x"ff";
new_byte      <= my_byte;
new_word      <= my_byte & my_byte; -- concatenation
my_int          <= my_natural;
my_natural    <= my_int ; -- WRONG
```

© All rights reserved to Abramov B.

Subtypes

```
subtype two_bits is std_logic_vector(1 downto 0);

signal a, b   : std_logic;

signal ab     : std_logic_vector(1 downto 0);

…………………………….

……………………………

ab <= two_bits'(a & b);
```



© All rights reserved to Abramov B.

Subtypes

Pre-defined subtypes:

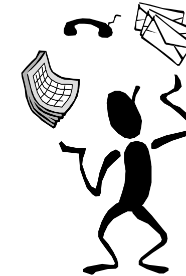
```
-- ( 0 to MAX_INTEGER)
subtype natural is integer range 0 to integer'high;
-- ( 1 to MAX_INTEGER)
subtype positive is integer range 1 to integer'high;

subtype delay_lengt is time range 0 fs to time'high;
```

For modeling and simulations only !!!

© All rights reserved to Abramov B.

Type conversions

Type conversions change an expression's type.



The syntax of a type conversion is

type_name(expression)

type_name - The name of a defined type.

The expression must evaluate to a value of a type that is convertible into type type_name.

- Type conversions can convert between integer types or between similar array types.
- Two array types are similar if they have the same length and have convertible or identical element types.
- Enumerated types are not convertible.

© All rights reserved to Abramov B.

Type conversions

Example for an Integer type conversion :

type small_int **is range** 0 **to** 10;

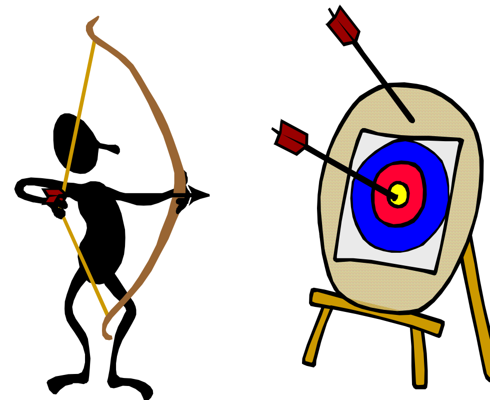
type big_int **is range** 0 **to** 20;

signal intermediate : small_int ;

signal final   : big_int;

intermediate <= 10;

final <= big_int(intermediate) + 2;



© All rights reserved to Abramov B.

Type conversions

- VHDL allows overloaded operators and conversions.

What conversion functions are needed?

- Signed & Unsigned (elements) <=> Std_Logic
- Signed & Unsigned <=> Std_Logic_Vector
- Signed & Unsigned <=> Integer
- Std_Logic_Vector <=> Integer

- VHDL Built-In Conversions
  - Automatic Type Conversion
  - Conversion by Type Casting
- Most Conversion functions are located in the Numeric_Std package.

© All rights reserved to Abramov B.

Type conversions

Most used functions for type conversions are:

conv_integer(arg : **std_logic_vector**) **return** **integer**;

conv_std_logic_vector(arg :**integer**, size :**integer**) **return** **std_logic_vector**;

to_bit(arg : **std_logic** ,xmap : bit=‘0’) **return** **bit**;

to_bit(arg : **std_ulogic** ,xmap : bit=‘0’) **return** **bit**;

to_bitvector(arg : **std_logic_vector** ,xmap : bit=‘0’) **return** **bit_vector**;

to_bitvector(arg : **std_ulogic_vector** , xmap : bit=‘0’) **return** **bit_vector**;

to_stdlogicvector(arg :**bit_vector**) **return** **std_logic_vector**;

to_stdlogicvector(arg : **std_ulogic_vector**) **return** **std_logic_vector**;

to_stdulogicvector(arg :**bit_vector**) **return** **std_ulogic_vector**;

to_stdulogicvector(arg : **std_logic_vector**) **return** **std_ulogic_vector**;

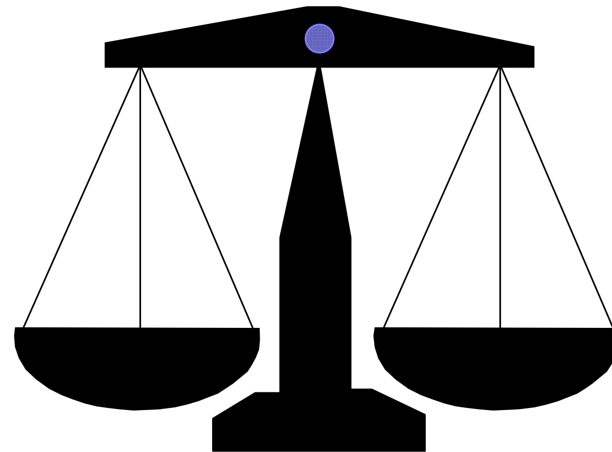
© All rights reserved to Abramov B.

Expressions and Operators

Expressions in VHDL are much like expressions in other programming languages. An expression is a formula combining primaries with operators.

Primaries include:

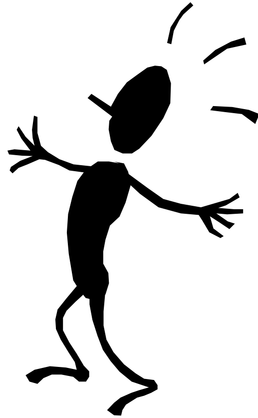
- names of objects
- literals
- function calls
- parenthesized expressions.



© All rights reserved to Abramov B.

Expressions and Operators

- **and**
- **or**
- **nand**
- **nor**
- **xor**
- **xnor**
- **not**



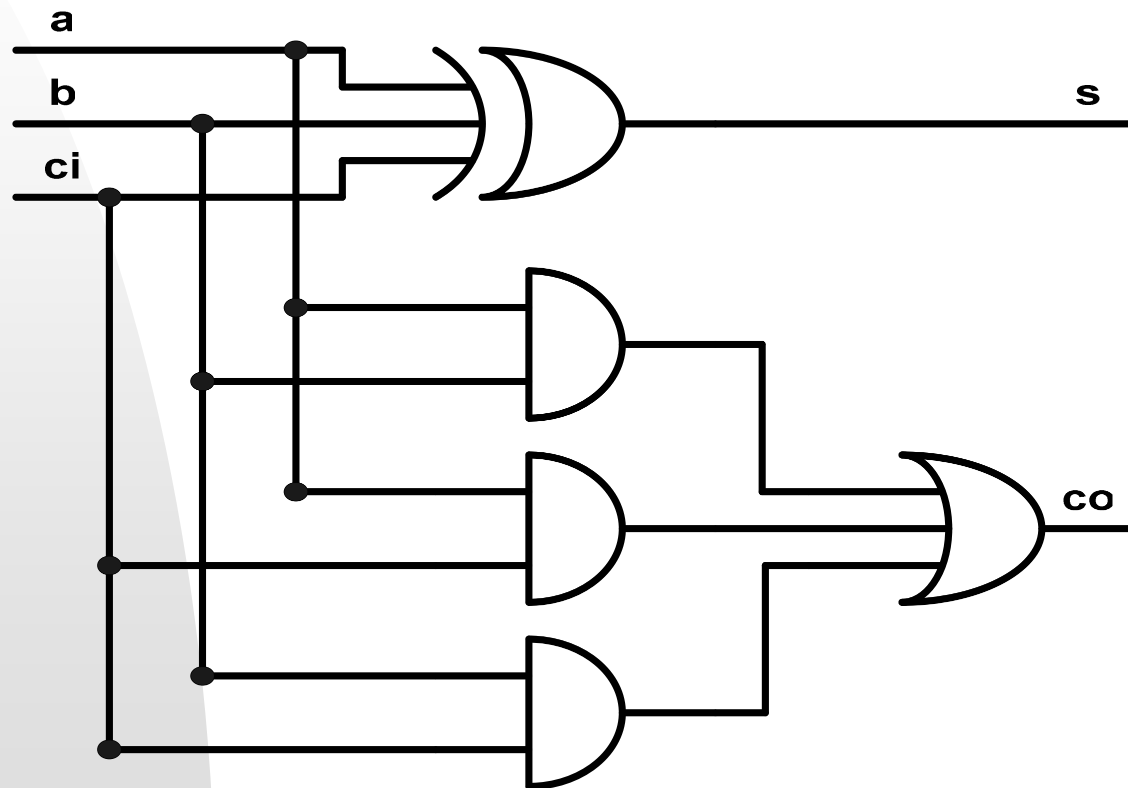
The above logical operators operate on values of types **bit**, **boolean**, **std_ulogic**, **std_logic** and also on one-dimensional arrays of these types.

For array operands, the operation is applied between corresponding elements of each array, yielding an array of the same length as the result.

© All rights reserved to Abramov B.

Expressions and Operators

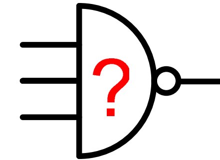
s <= a **xor** b **xor** ci;
co <= (a **and** b) **or** (a **and** ci) **or** (b **and** ci);
The logical schematic of the above description is:



© All rights reserved to Abramov B.

Expressions and Operators

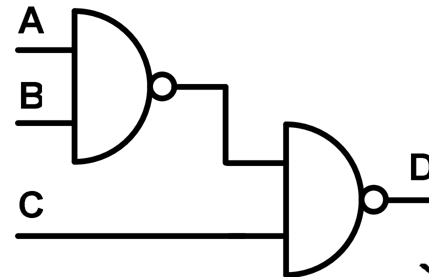
Problem : Describe a - 3 input NAND gate in VHDL.



Suggested solution :

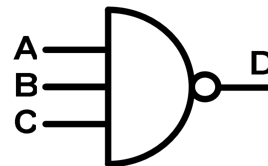
D <= A **nand** B **nand** C; -- Syntax wrong !!!

The suggested solution will produce the following circuit :



Correct solution :

D <= **not** (A **and** B **and** C); -- Correct

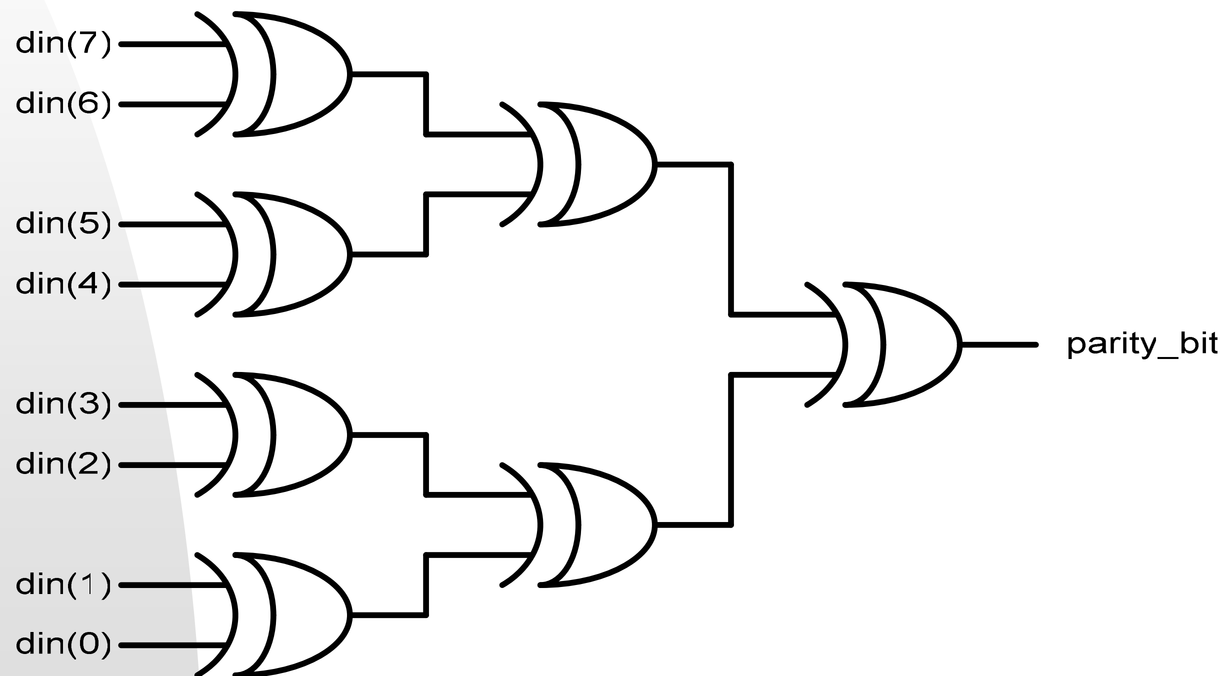


© All rights reserved to Abramov B.

Expressions and Operators

Hierarchy with parentheses:

parity_bit <= ((din(7) **xor** din(6)) **xor** (din(5) **xor** din(4))) **xor**
((din(3) **xor** din(2)) **xor** (din(1) **xor** din(0)));



© All rights reserved to Abramov B.

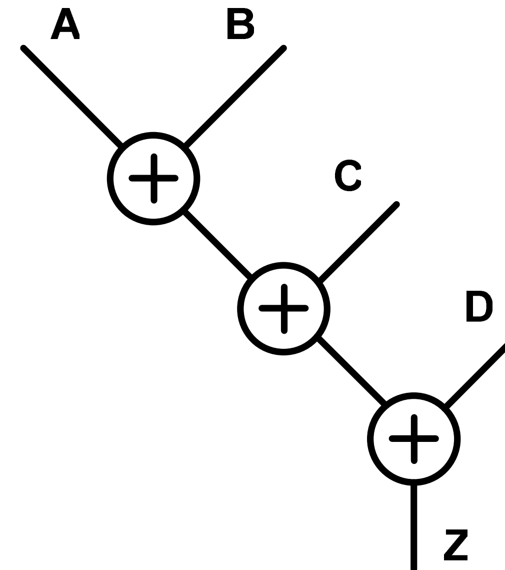
Expressions and Operators

Simple Arithmetic Expression

Z <= A + B + C + D;

The parser performs each addition in order, as though parentheses were placed within the expression as follows:

Z <= ((A + B) + C) + D;



The compiler constructs the expression tree.

© All rights reserved to Abramov B.

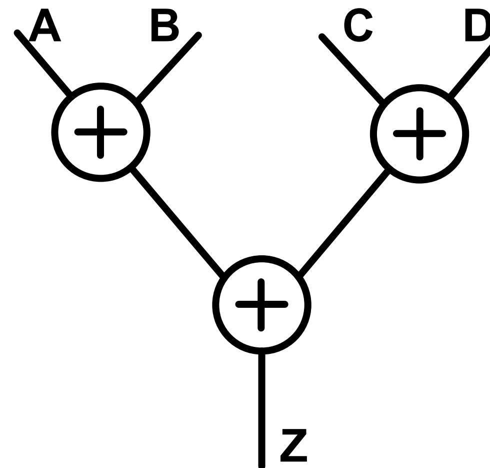
Expressions and Operators

If the arrival times of all the signals are the same,

the length of the critical path of the expression equals three adder delays.

The critical path delay can be reduced to two adder delays if you insert parentheses as follows:

Z <= (A + B) + (C + D);



Balanced Adder Tree (Same Arrival Times for All Signals)

© All rights reserved to Abramov B.

Expressions and Operators

The relational operators :

- =
- /=
- <
- <=
- >
- >=

must have both operands of the same type, and all yield boolean results.

The equality operators (= and /=) can have operands of any type.
For composite types, two values are equal if all of their corresponding elements are equal.

The remaining operators must have operands which are scalar types or one-dimensional arrays of discrete types.

© All rights reserved to Abramov B.

Expressions and Operators

The sign operators **(**+ and **-)** and the addition (+) and subtraction (-) operators have their usual meaning on numeric operands.

```
entity adder is
port(
        a : in  integer range 31 downto 0;
        b : in  integer range 31 downto 0;
        c : out integer range 63 downto 0
    );
end entity adder;

architecture arc_adder of adder is
begin
        c <= a + b;
end architecture arc_adder;
```

© All rights reserved to Abramov B.

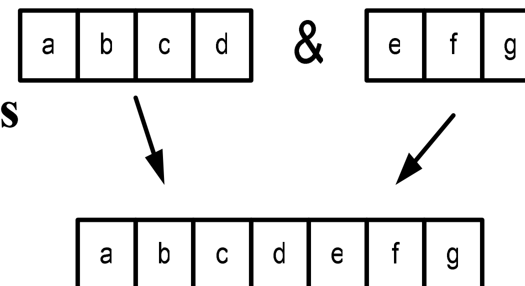
Expressions and Operators

The concatenation operator (&) operates on one-dimensional arrays to form a new array with the contents of the right operand following the contents of the left operand.

It can also concatenate a single new element to an array, or two single bit elements to form an array.

The concatenation operator is most commonly used with bit strings.

```
entity bus_conc  is
port( a : in   std_logic_vector (7   downto 0);
        b : in   std_logic_vector (7   downto 0);
        c : out std_logic_vector (15 downto 0));
end entity bus_conc ;

architecture arc_ bus_conc of bus_conc is
begin
      c <= a & b;
end architecture bus_conc ;
```



© All rights reserved to Abramov B.

Expressions and Operators

The multiplication (*) and division (/) operators work on ***integer***, floating point and physical types**.**

The modulus (**mod**) and remainder (**rem**) operators only work on integer types.
The absolute value (**abs**) operator works on any numeric type.

The exponentiation (**) operator can have an integer or floating point left operand, but must have an integer right operand.
A negative right operand is only allowed if the left operand is a floating point number.

© All rights reserved to Abramov B.

Expressions and Operators

Example :

```
entity alarm is
    port(
        temp_level     : in  integer;
        critical _level : in  integer;
        alarm_sig      : out boolean );
end entity alarm;

architecture arc_ alarm of alarm is
begin
-- the temperature should be in the range of  –critical_level to +critical_level
    alarm_sig <= ( (abs temp_level) >= critical_level);
end architecture arc_ alarm;
```

© All rights reserved to Abramov B.

Expressions and Operators

The mod function is defined as the amount by which a number exceeds the largest integer multiple of the divisor that is not greater than that number. With a positive number the multiple we subtract is SMALLER in absolute value.

With negative numbers, we subtract a number with a LARGER absolute value

A **mod** B = A – (int(A/B)) *B -- when A and B have the same sign

A **mod** B = A + (ceil(|A/B|)*B -- when A and B have different sign

A **mod** B = A – B * N (Where N is defined as an integer.)

- ✓ the sign of (A **mod** B) is the same as the sign of B
- ✓ **abs** (A **mod** B) < **abs** (B)

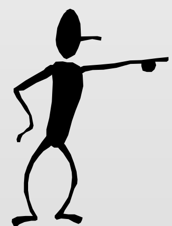
c <= (c + 1) **mod** 16; -- c would never be more then 15

Note : 7 mod 4 → 7 – (1)*(4) = 7 – 4 = 3

7 mod (-4) → 7 + (2)*(-4) = 7 + (-8) = -1

(-7) mod 4 → (-7) + (2)*(4) = (-7) + 8 = 1

(-7) mod (-4) → (-7) – (1)*(-4) = (-7) – (-4) = -3



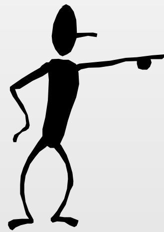
For synthesis the modulo operator is supported only when the right operand is power of 2. (2**n)

© All rights reserved to Abramov B.

Expressions and Operators

The remainder operation is defined such that the relation:
a = (a / b) * b + (a **rem** b) or (a **rem** b) = a – (a / b) * b
and
abs (a **rem** b) < **abs** (b),

The result of the **rem** operator has the sign of its first operand , while the result of the **mod** operators has the sign of the second operand.



6 **rem** 4	--	2
4 **rem** 6	--	4
6 **rem** (-4)	--	-2
(-6) **rem** 4	--	2

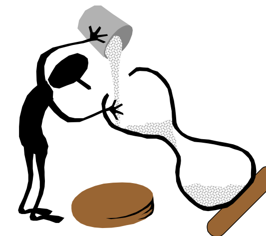
For synthesis the remainder operator is supported ONLY when the right operand is 2**n

© All rights reserved to Abramov B.

Expressions and Operators

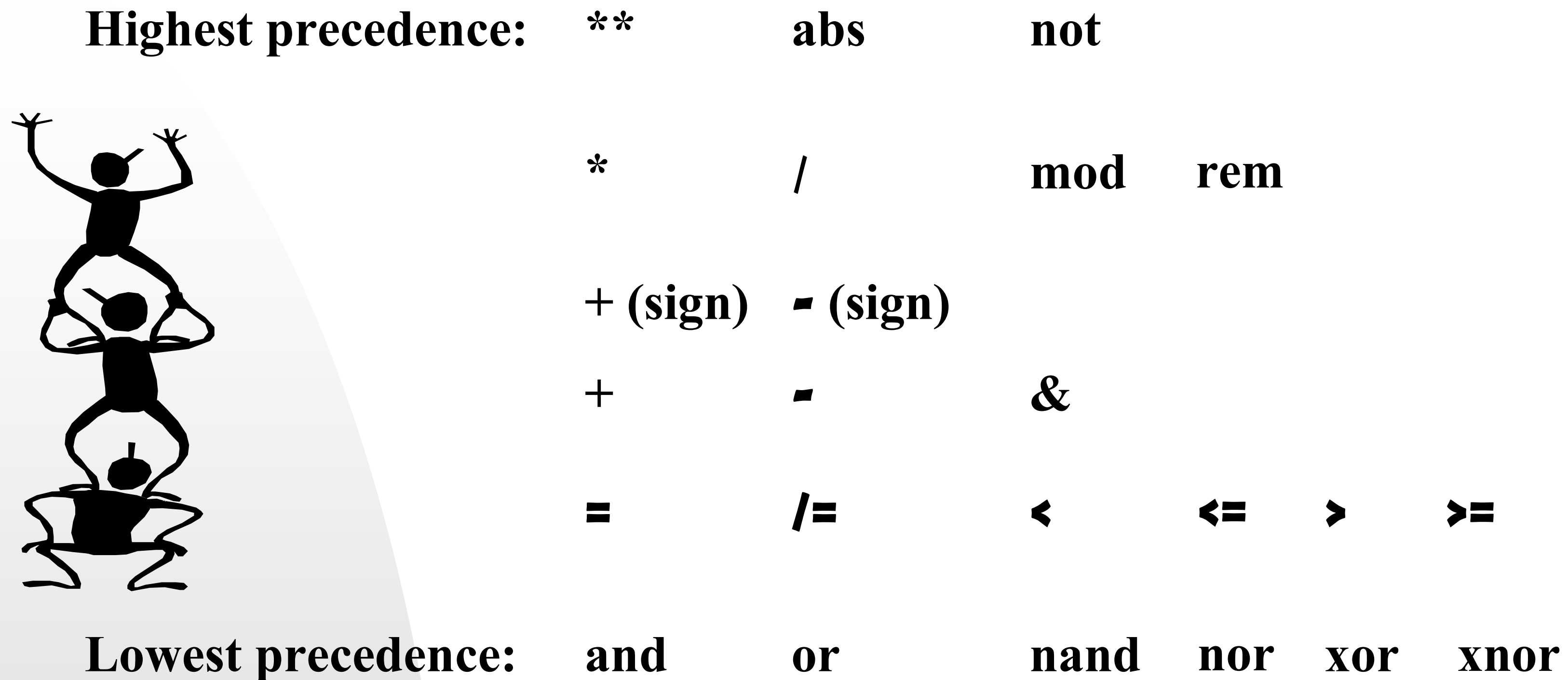
Sample of usage of power operator by integer range declaration :

```
entity mul is
port(
        a   : in  integer range ((2**8) – 1)  downto 0;
        b   : in  integer range ((2**8) – 1)  downto 0;
        res : out integer range ((2**16) – 1) downto 0
    );
end entity mul;

architecture arc_mul of mul is
begin
        res <= a * b;
end architecture arc_mul;
```



© All rights reserved to Abramov B.

Expressions and Operators



Highest precedence:	**	abs	not			
	*	/	mod	rem		
	+ (sign)	- (sign)				
	+	-	&			
	=	/=	<	<=	>	>=
Lowest precedence:	and	or	nand	nor	xor	xnor

Note : Precedence on equal level operators is done left to right.

© All rights reserved to Abramov B.

Expressions and Operators

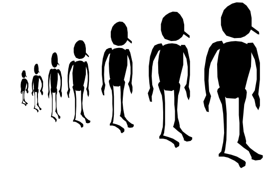
VHDL-93 has a wide set of shift operators.
The table bellow list the shift operators:

Name	Operation
sll	Shift Left Logical – '0' are pushed from right
srl	Shift Right Logical – '0' are pushed from left
sla	Shift Left Arithmetic – Right-most bit is pushed from right
sra	Shift Right Arithmetic – Left-most bit is pushed from left
rol	Rotate Left
ror	Rotate Right

© All rights reserved to Abramov B.

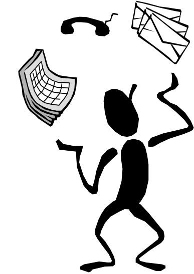
Expressions and Operators

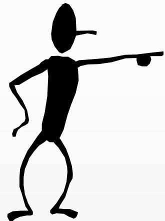
The shift operators are defined for the one-dimensional array with the elements of type ***bit*** .
For the shift operator the left operand L is an array and the right operand R is an integer.
The right operand R represents the number of positions the left operand L should be shifted.
As the result of shifting, the value of the same type as the left operand is returned.



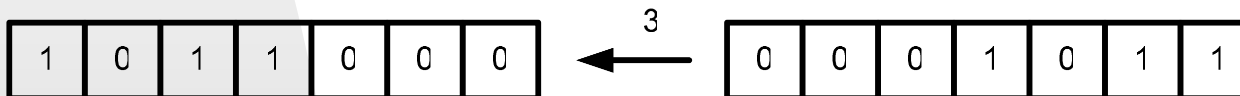
© All rights reserved to Abramov B.

Expressions and Operators





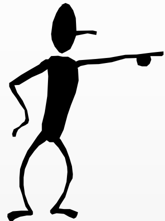
```
signal source : bit_vector(3 downto 0);
signal dest   : bit_vector(3 downto 0);
...............
source <= "1011";
 -- On Shift Left Logical (sll) and Shift Right logical (srl)
 -- '0's will be pushed in.
dest <= source  sll   1;    -- dest = "0110"  - '0' is inserted
dest <= source  srl   1;    -- dest = "0101"  - '0' is inserted
```



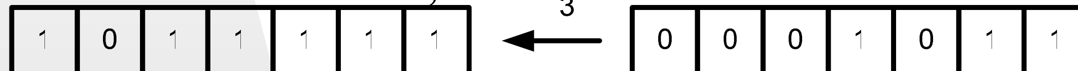

```
 -- On a negative shift value the direction will be the opposite
dest <= source  sll –3;    -- dest <= source srl 3 ( "0001")
```

© All rights reserved to Abramov B.

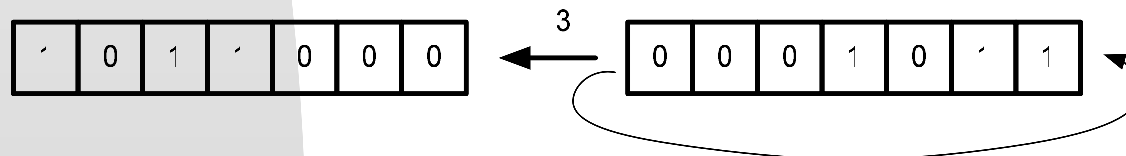
Expressions and Operators



signal source : bit_vector(3 **downto** 0);
signal dest : bit_vector(3 **downto** 0);
……………
source <= ”1011”;
-- On Shift Left Arithmetic (sla) and Shift Right Arithmetic (sra)
-- The inserted bit will be the same as the bit pushed from the
-- edge to the center of the word
dest <= source **sla** 1; -- dest = ”0111” -- ‘1’ is inserted
dest <= source **sra** 1; -- dest = ”1101” -- ‘1’ is inserted



-- The rotate operation cycles the dropped bit back to the register
dest <= source **rol** 1; -- des t= ”0111”
dest <= source **ror** 1; -- dest = ”1101”



© All rights reserved to Abramov B.

Expressions and Operators

The shift operators are defined for one-dimensional array with elements of type ***bit*** (***bit_vector***)

However they can be used for ***std_logic_vector***, but only along with type conversion function.

```
signal  source : std_logic_vector(3 downto 0);
signal  dest   : std_logic_vector(3 downto 0);
……………

dest <= to_stdlogicvector(to_bitvector(source) sll 2);
```

© All rights reserved to Abramov B.

Expressions and Operators

Sources and target sizes:

Expression	Size of Y
Y <= A ;	A'Length
Y <= A and B ;	A'Length = B'Length
Y <= A > B ;	Boolean
Y <= A + B ;	Maximum (A'Length, B'Length)
Y <= A + 10 ;	A'Length
Y <= A * B ;	A'Length + B'Length

© All rights reserved to Abramov B.

Expressions and Operators

VHDL strictly demands conformity of the size of operands.

This excessive severity can sometimes be an obstacle, for example at realization of an addition operation.

In an addition operation the size of the result might be one bit bigger than the size of the bigger operand. But the language defines the result size to be as the size of the bigger operand and not more than that.

How it is possible to overcome this contradiction?

One of possible variants :

```
signal a, b : std_logic_vector(n downto 0);
signal sum : std_logic_vector((n+1) downto 0);
sum <= ('0' & a) + ('0' & b);
```

© All rights reserved to Abramov B.

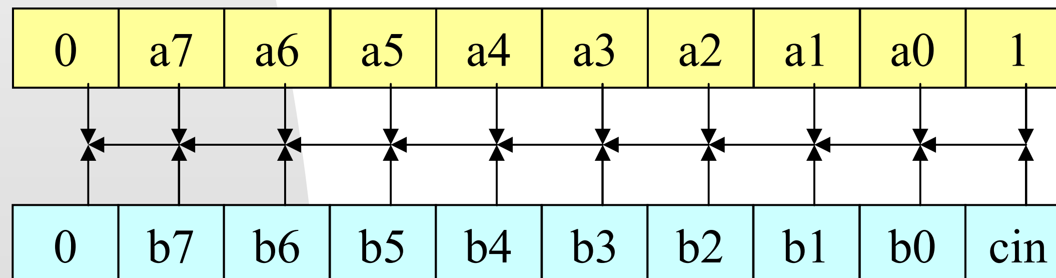
Expressions and Operators

Problem : Perform the next operation → A + B + cin

Where A and B are two buses of 8 bit width and cin is a carry input bit.

Solution:

```
signal temp_a, temp_b, temp_sum : std_logic_vector(9 downto 0);
signal sum                      : std_logic_vector(8 downto 0);
temp_a <= '0' & a & '1';  temp_b <= '0' & b & cin;
temp_sum <= temp_a + temp_b;
sum <= temp_sum(9 downto 1);
```

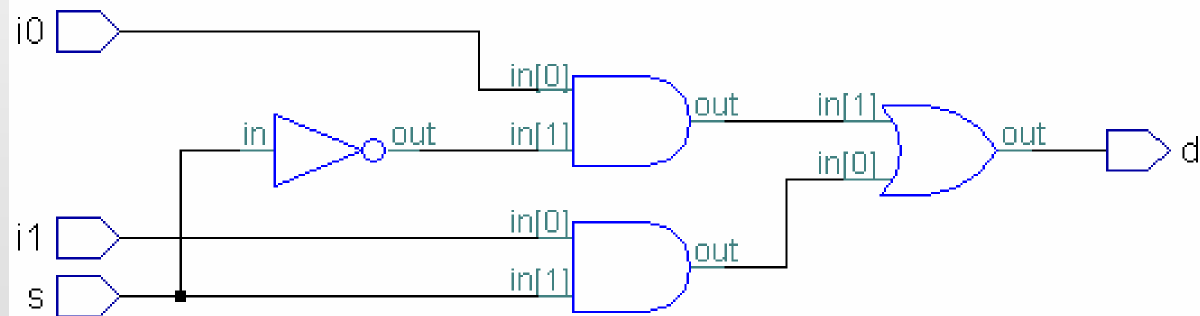


Where each node of the scheme is a relevant bit of temp_sum

© All rights reserved to Abramov B.

Self Work

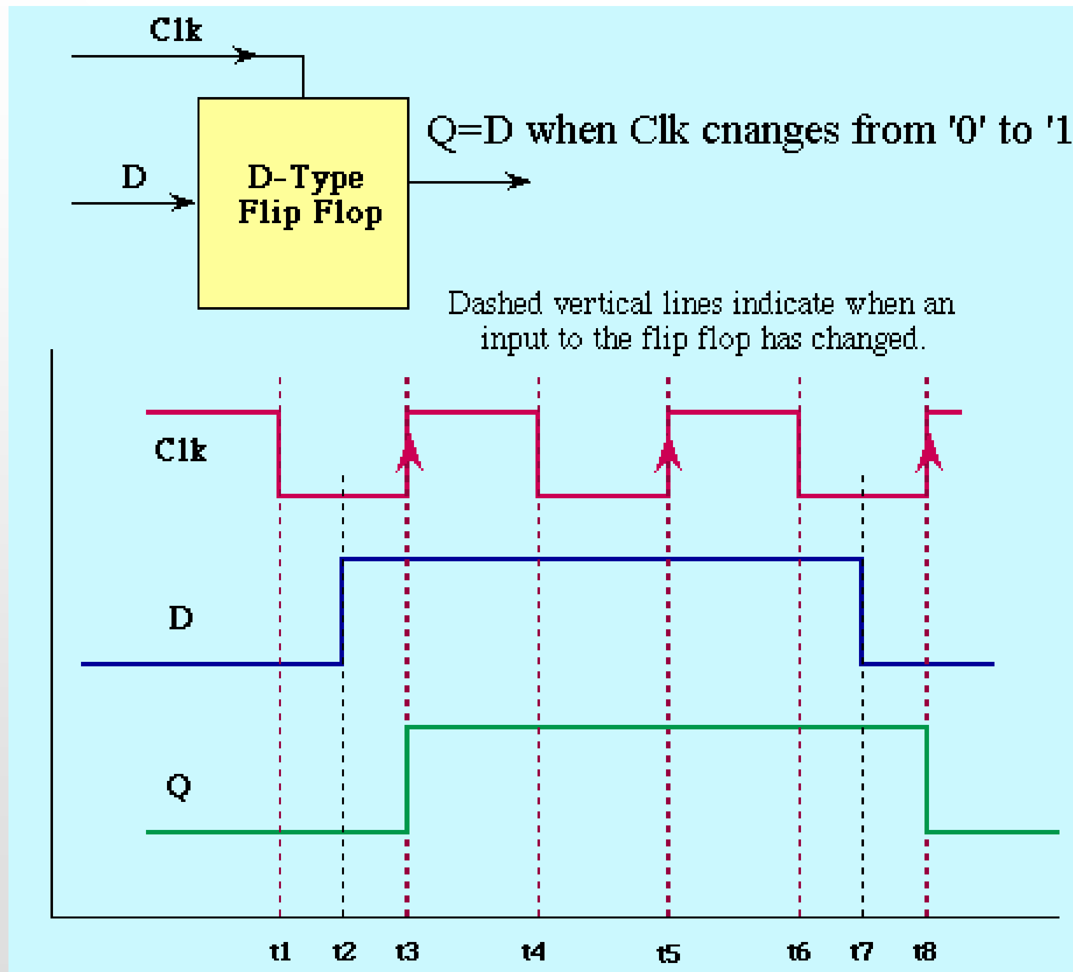
Describe mux 2x1 on gate level according to followed scheme:



© All rights reserved to Abramov B.

Self Work

Implement the rising-edge sensitive DFF from mux 2x1 components



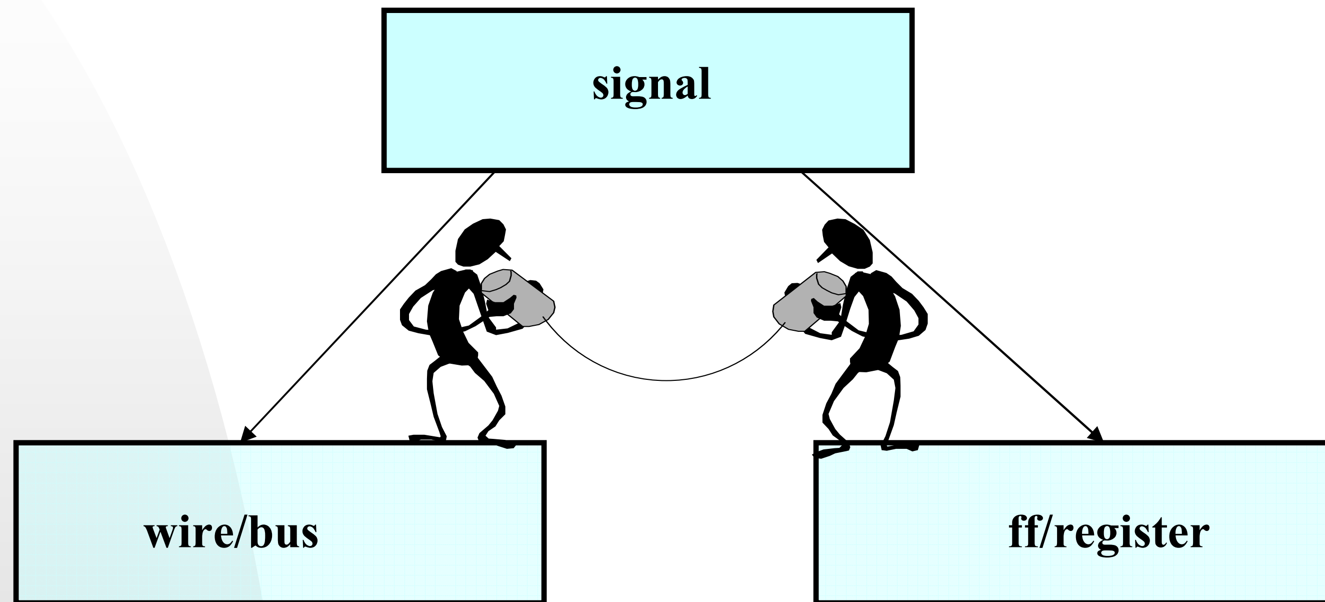
© All rights reserved to Abramov B.

Topic Three

- Signal 151
- Constant.......................161
- Process.......................163
- Combinatorial Process..... 169
- Synchronous Process.......184
- Sequential Statements......204
- NULL Statements............240
- Concurrent Statements......242
- Delay Mechanism............244
- Test Bench....................253
- Attributes......................270
- Wait Statements..............297

© All rights reserved to Abramov B.

Signal

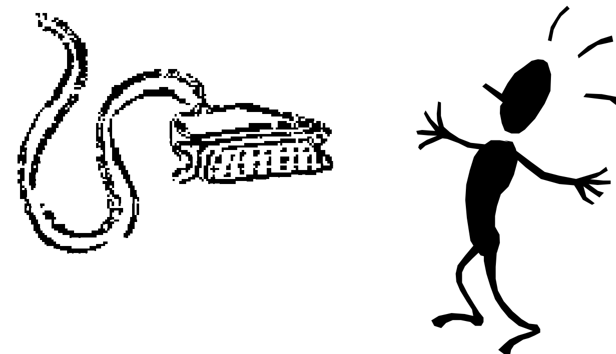


© All rights reserved to Abramov B.

Signal

- Signals are the primary object describing a hardware system and are equivalent to “wires”.
- Represent communication channels among concurrent statements of system’s specification.
- Signals and associated mechanisms of VHDL

(signal assignment statements, resolution function, delays, etc.) are used to model inherent hardware features such as concurrency, inertial characters of signal propagation, buses with multiple driving sources, etc.



© All rights reserved to Abramov B.

Signal

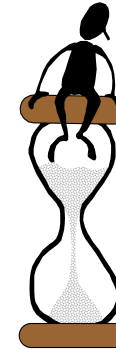
Each signal has a history of values and may have multiple drivers, each of which has a current value and projected future values.

Signal can be explicitly declared in the declarative part of:

- Package declaration; signals declared in a package are visible in every entity that is using the package.
- Architecture: signal is visible inside the architecture only.
- Block: the scope of such signals is limited to the block itself;
- Subprogram (function and procedure);

© All rights reserved to Abramov B.

Signal



The signals can be classified as either internal or external.

External signals are signals that connect the system to the outside world, they form the system's interface (ports).

Internal signals, which are not visible from the outside, are completely embedded inside the system and are part of its internal architecture, providing signals between internal circuits.

© All rights reserved to Abramov B.

Signal

All signal parameters are accessible by means of signal attributes.

A signal is another form of an object in VHDL.
All objects and expressions in VHDL are strongly typed.
This means that all objects are of a defined type and issues an error message if there is a type mismatch.

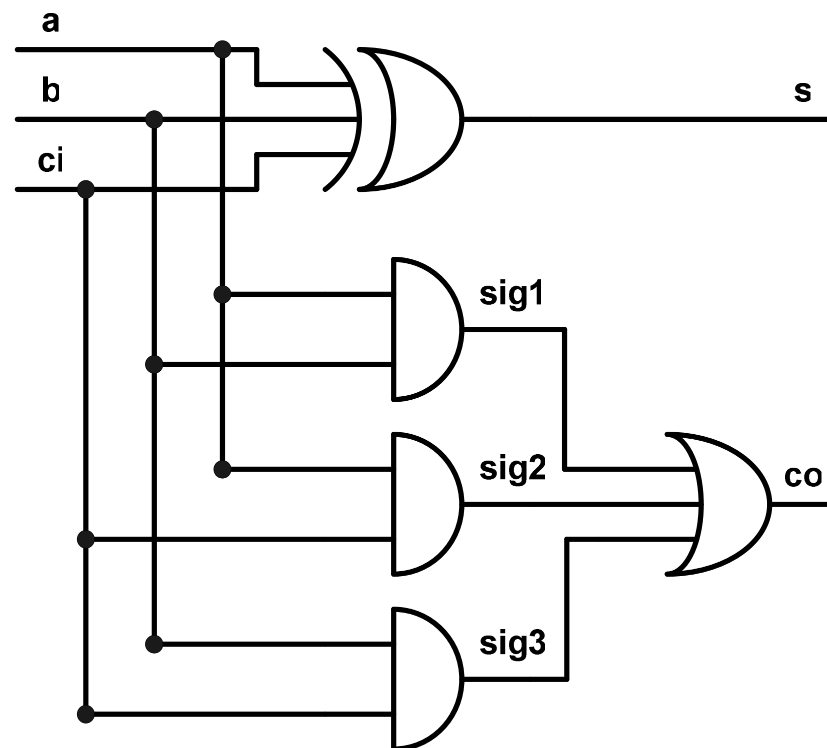
For example, you cannot assign an signal of type **integer** to a **std_logic**.

© All rights reserved to Abramov B.

Signal

The architecture with in signal declaration:

```
architecture arc_full_adder of full_adder is
    -- signal declaration
    signal sig1 : std_logic;
    signal sig2 : std_logic;
    signal sig3 : std_logic;
begin
    s   <= a xor b xor ci;
    co <= sig1 or sig2 or sig3 ;
    -- signal assignment
    sig1  <= a and b;
    sig2  <= a and ci;
    sig3  <= b and ci;
end architecture arc_full_adder;
```



© All rights reserved to Abramov B.

Signal

- ✓Signal assignment statement can appear inside a process or directly in an architecture
- ✓Sequential signal assignment or concurrent signal assignment can be used
- ✓All signal assignment can be delayed
- ✓Signal may be initialized during signals declaration (ignored by synthesis)
- ✓Only resolved types signal supports multiple drivers
- ✓All signal assignment can be labeled for improved readability
- ✓ Ports of entity as such signals

© All rights reserved to Abramov B.

Signal (cont)

```
signal a, b, c : bit := '0';

      a <= '1';
      ……
      a <= c and b;
```

Illegal assignment, a is unresolved signal

```
signal  rslv_sig : std_logic;


rslv_sig <= a  when (os = '1') else 'Z';

rslv_sig <= b  when (os = '0') else 'Z';
```

Legal VHDL

© All rights reserved to Abramov B.

Signal

Signal assignment for behavioral description.

Syntax:

Signal_name <= (value_expression [after time expession]), (, …) ;

Examples :

Y <= **not** ((a **and** b) **or** c) **after** 10 ns;

rst_n <= ‘1’ , ‘0’ **after** 10 ns, ‘1’ **after** 50 ns;

Acceptable VHDL, but delay is ignored by synthesis tools

Not supports by synthesis tools

© All rights reserved to Abramov B.

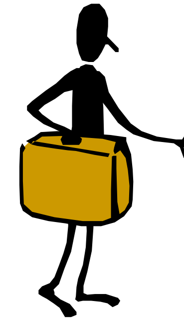
Constant

Used for holding a value that can not be changed during design description.

- Constants value is assigned during declaration
- Used for readability and ease of design modification reasons.

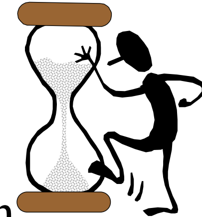
Syntax:

constant constant_name : **type** := value;



© All rights reserved to Abramov B.

Constant



```
constant byte_width      : integer := 8;
constant num_of_bytes    : integer := 4;
constant num_of_bits     : integer := num_of_bytes * bus_width;
signal my_dword          : std_logic_vector( (num_of_bits – 1) downto 0);
signal my_bus            : std_logic_vector( (byte_width –1) downto 0);
signal my_sig            : std_logic_vector(15 downto 0) := x”f0f0”;!!!
constant all_ones        : std_logic_vector(15 downto 0) := x”ffff”;
constant VCC             : std_logic := ‘1’;
constant GND             : std_logic := ‘0’;

-----------------------------------------------------------------------------

my_sig <= all_ones; -- the constant used to set of default value
```

© All rights reserved to Abramov B.