Digital Design with VHDL

Language Structure

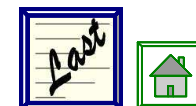
By Benjamin Abramov

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

Language Structure

- Lexical Elements......................................3
- Meta Comments..8
- Entity...9
- Port..13
- Port mode 14
- Types 18
- Architecture 35
- Signal & Components.............................42
- Test Bench...54

© All rights reserved to Abramov B.



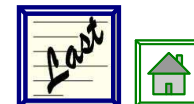
Lexical Elements

Naming syntax – **Applies to all user defined names**.

All must follow the following rules :

- May contain one or more of the following characters up to a maximum of 32 characters:
  - Alphabetic letters – ‘A’ to ‘Z’ and ‘a’ to ‘z’ .
  - Digits – ‘0’ to ‘9’.
  - Underscore (underline) – **‘_’** .
- Must start with an alphanumeric letter.
- May not include two successive underscore characters.
- May not end with an underscore.

© All rights reserved to Abramov B.



Lexical Elements

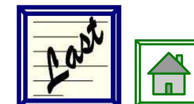
__Valid identifiers examples:__

Count_enable
Shift_reg_8bit
Four_biT_CounTer
write_en_n

__Invalid identifiers examples:__

8bit_shift_reg -- start with digit
count__enable -- two successive underlines
Four_biT's_CounTer -- contains an illegal character - (')
write_en_ -- ends with an underscore

© All rights reserved to Abramov B.



Lexical Elements

Special symbols:

VHDL uses a number of special symbols to denote operators and operations precedence, command and construct delimiters and punctuation. One character symbols:

“ # & ‘ ( ) * + - , . / : ; < = > [ ] |

Two characters symbols must be typed next to each other, with no intervening space.

<= => ** := /= >= <= <> --

VHDL is not a case sensitive language!

“a” and “A”, “deCOdeR” and “DEcoder” are the same names.

© All rights reserved to Abramov B.

Lexical Elements

Comments:

A comment can be added to a line by writing two dashes together, followed by the comment text.

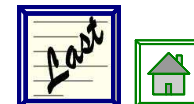
There is no block commenting (like in C for example).

Comment extends from the point it is marked to the end of the line. No code can be written on the line from the point where a comment was written

For example:

c <= a + b -- this is example of comment for the statement c <= b + a;

© All rights reserved to Abramov B.



Lexical Elements

Reserved words:

abs
access
after
alias
all
and
architecture
array
assert
attribute
begin
block
body
buffer
bus
case
component
configuration
constant

disconnected
downto
else
elsif
end
entity
exit
file
for
function
generate
generic
group
guarded
if
impure
in
inertial
inout
is

label
library
linkage
literal
loop
map
mod
nand
new
next
nor
not
null
of
on
open
or
others
out

package
port
postponed
procedure
process
protected
pure
range
record
register
reject
rem
report
return
rol
ror
select
severity
shared
signal

sla
sll
sra
srl
subtype
then
to
transport
type
unaffected
units
until
use
variable
wait
when
while
with
xnor
xor

© All rights reserved to Abramov B.

Meta-comments

The meta-comments, are compiler directed comments.

-- TRANSLATE_OFF

and

-- TRANSLATE_ON

The above compiler commands cause the synthesis tool to ignore the code encapsulated between them. The simulation compiler will not ignore these lines and it will process them.

Meta-comments are mostly compiler unique.

The designer should consult the tool being used in order to write the appropriate Meta-command.

© All rights reserved to Abramov B.

Entity

The first basic building block in **VHDL** is the **entity**.

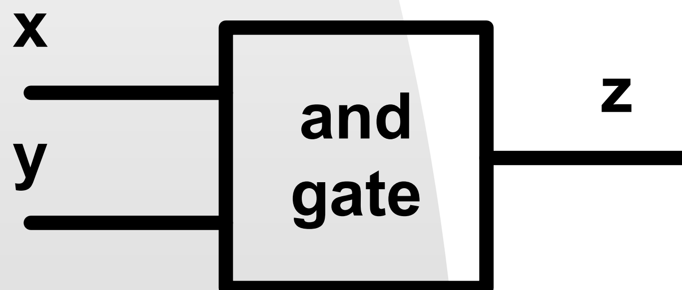
Entity describes the outer package of the logic unit. The **entity** part provides system interface specification. In simple electronic view an entity can be described as a package of an IC. The entity provides the interface between the design and the outer world.

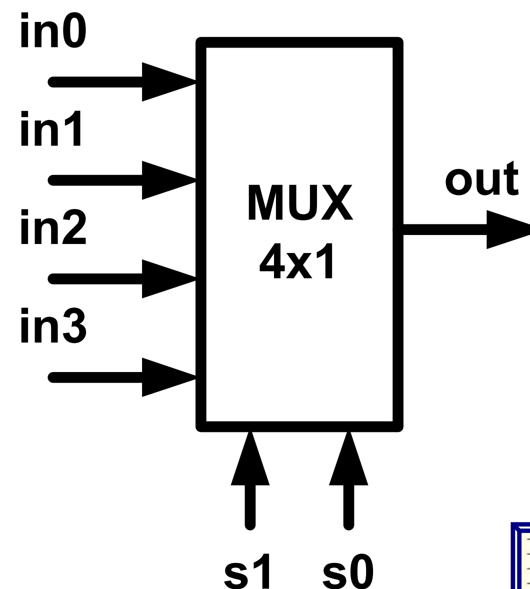
© All rights reserved to Abramov B.

Entity

Entities in VHDL can describe systems with different complexity levels:

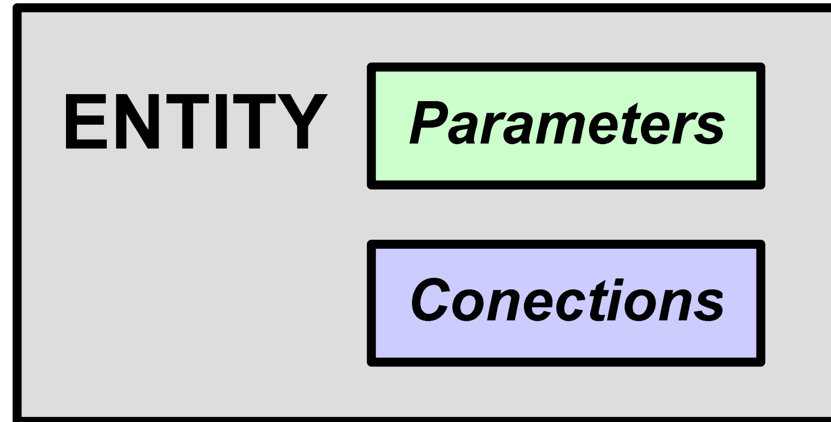
- **Gates**
- **Discrete components (multiplexor, decoder …).**
- **Complex digital designs such as controllers and processors (Power PC, Pentium IV, 8051).**
- **A full system (PCI card, motherboard, …).**
- **Each of these will be described by their outer world interface:**





© All rights reserved to Abramov B.

Entity



The entity part is generally comprised of two elements:

- **Parameters of the system as seen from the outside (such the bus width, maximum clock frequency…) -** ***generics***
- **Connections that are transferring information to and from the system (system’s inputs and outputs) -** ***ports***

© All rights reserved to Abramov B.

Entity

The Entity block syntax is shown below :
Ports and ***Generics*** are declared within the Entity.

Syntax:

entity entity_name **is**
generic(generic_list);
port(port_list);
end entity entity_name;

© All rights reserved to Abramov B.

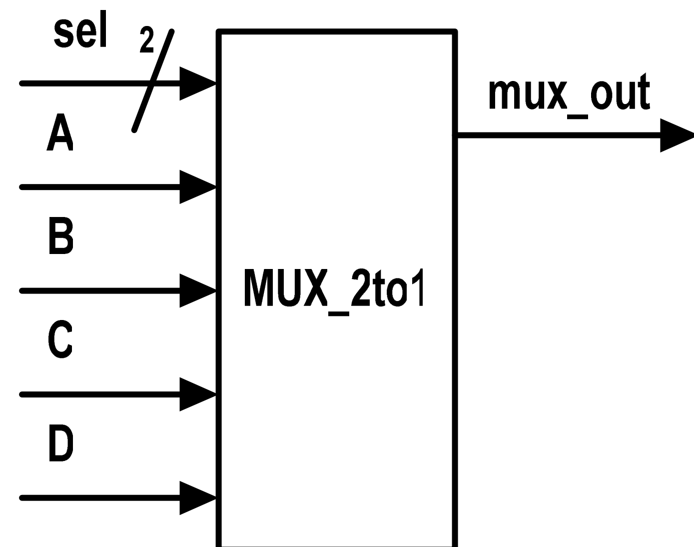
Entity

The **port** list must define:

- **Name** – must be unique
- **Mode** – direction of the port
- **Type** – values it can contain

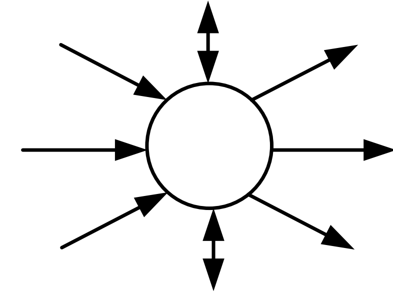
Example :

```
entity  mux_2to1 is
  port( A, B : in  bit;
        C, D : in  bit;
        sel     : in  bit_vector(1 downto 0);
        mux_out : out  bit);
end entity mux_2to1;
```



© All rights reserved to Abramov B.

Entity



Port mode and read/write status :

Mode	Read	Write
in	yes	no
out	no	yes
buffer	yes	yes
inout	yes	yes

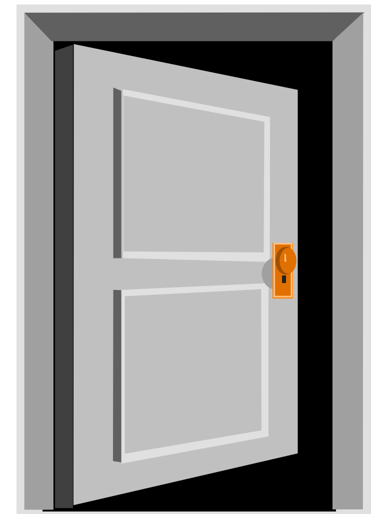
!!! NOTE : By default when no direction is declared for a port, the port is defined by default to be of mode **IN**.

© All rights reserved to Abramov B.

Entity

in, out modes :

- **IN – unidirectional port used as an input to the design.**
    - **May not be assigned within the design.**
- **OUT – unidirectional port used as an output from the design.**
    - **Must be driven a value within the design.**
    - **May not be read/sampled within the design.**

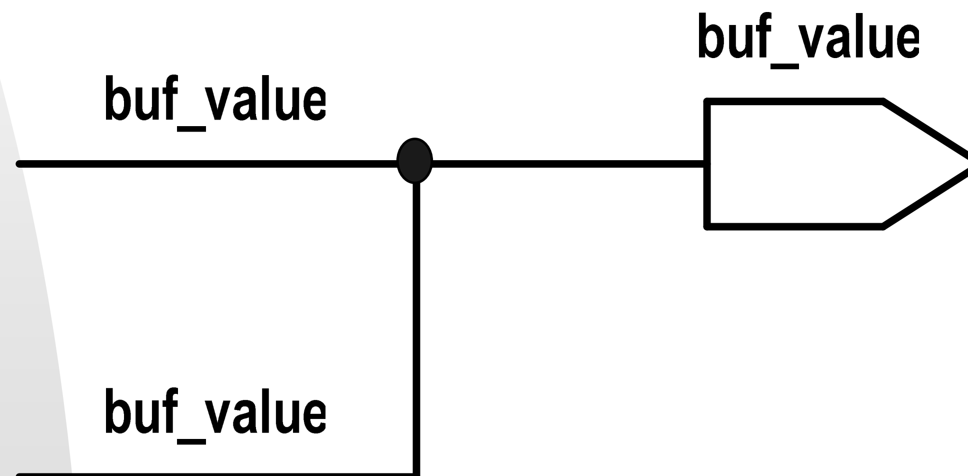


© All rights reserved to Abramov B.

Entity

buffer mode:

- **BUFER – An output port with a feedback drive to the design.**
  - **Must be driven with a value within the design.**
  - **May be read/sampled in the design.**



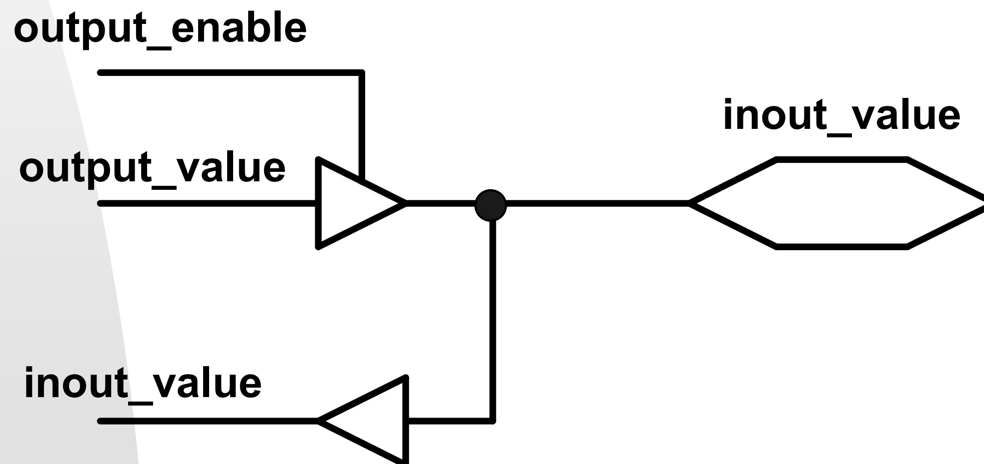
© All rights reserved to Abramov B.

Entity

inout mode:

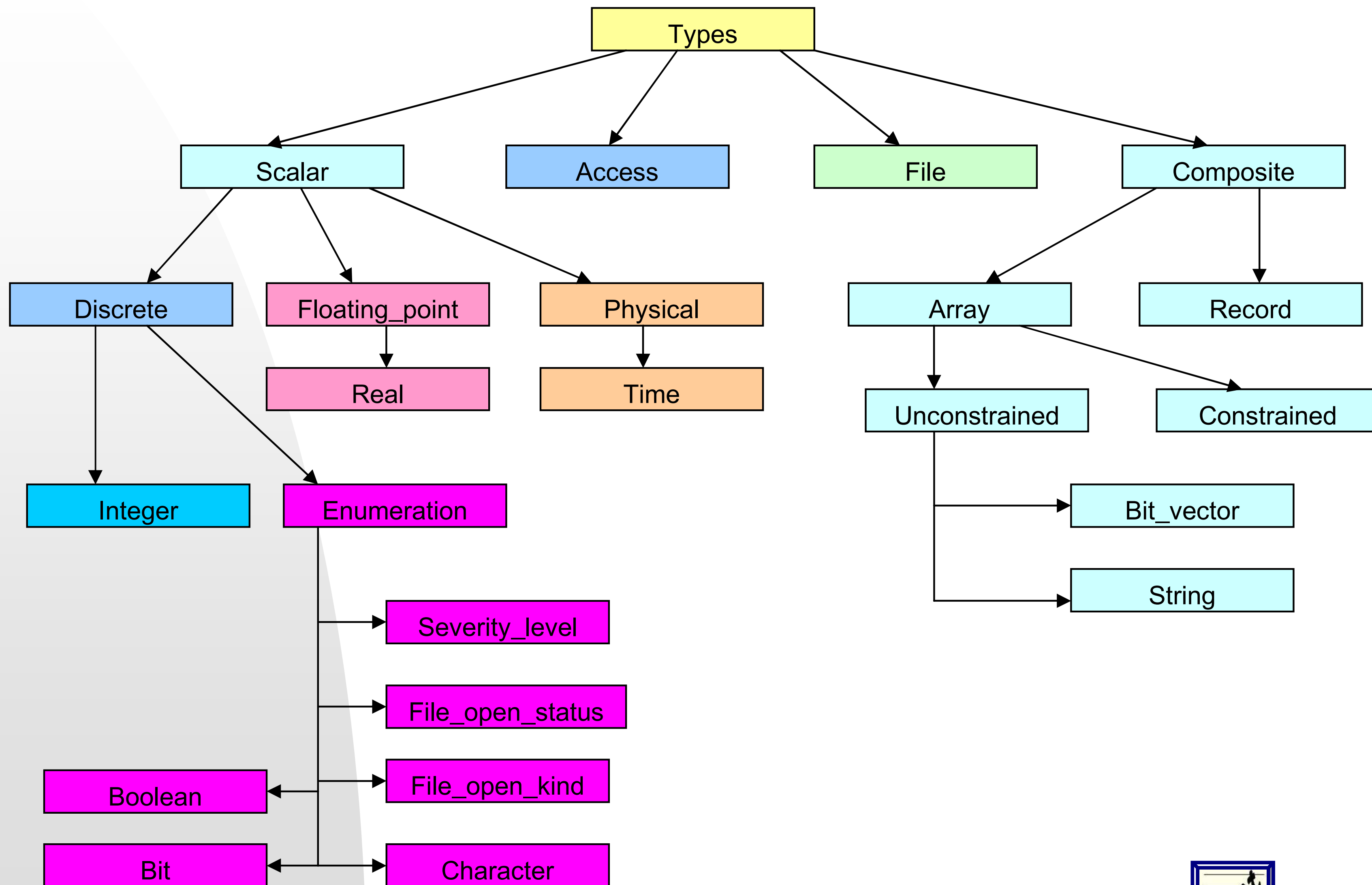
- **INOUT – A bidirectional port used as input and output.**
  - **May be read/sampled and written within the design.**

!!! It is the designers responsibility to avoid contention on the line.



© All rights reserved to Abramov B.

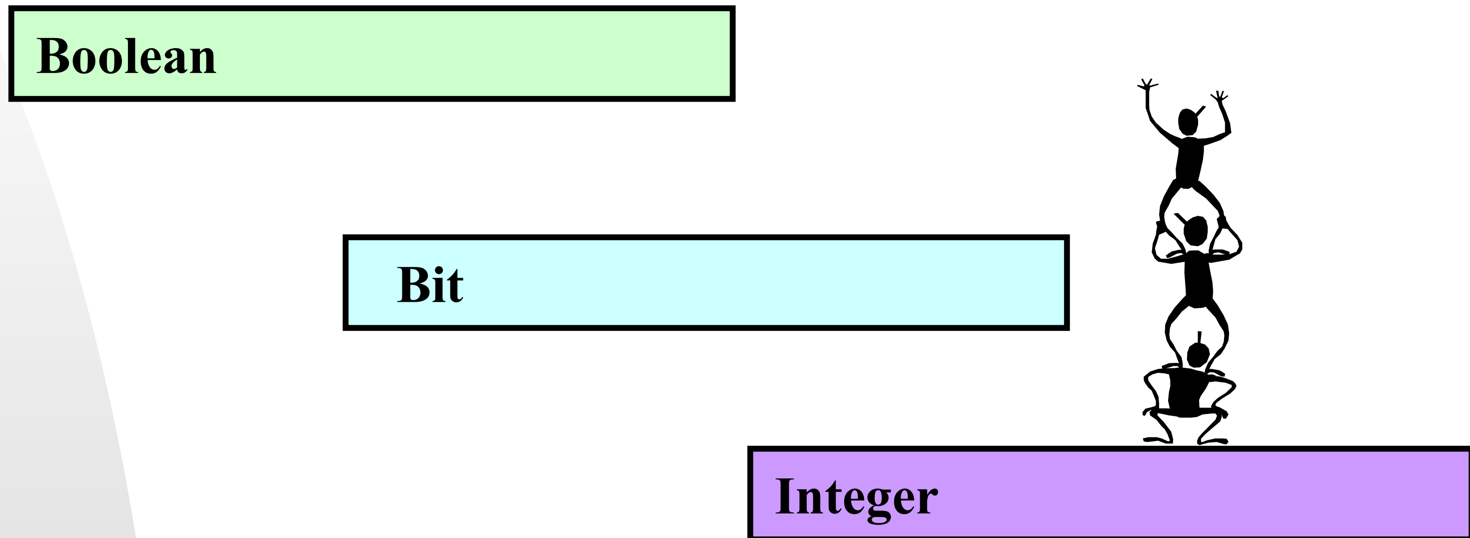
VHDL types hierarchy



© All rights reserved to Abramov B.

Basic Types

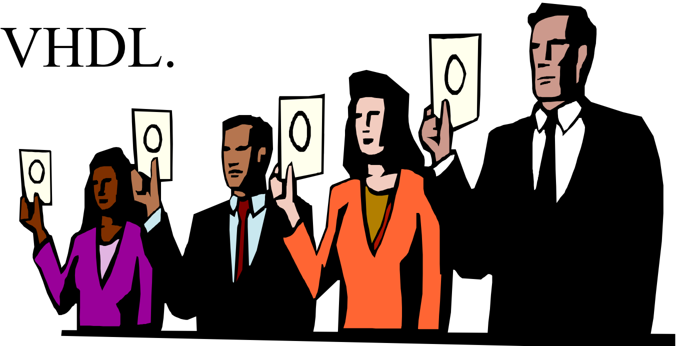
Three basic predefined types :



© All rights reserved to Abramov B.

Boolean

Boolean is a predefined **enumeration** type in VHDL.
Boolean type is defined as:



type boolean is (false, true);

The relational operators, equality (“=”) and inequality (“/=”), can be applied to operands of any types, including composite types.
The result of a relational operation will be of boolean type **true** or **false**.

For example:

10 = 10;	123 = 234;
“010” = ”010”;	“abc” /= “bcd”;
3 ns = 3 ns;	3 ns = 3 ps;

The above all yield the value true. ***The above all yield the value false.***

© All rights reserved to Abramov B.

Boolean

- The following **Relational** operators are applied only to ordered types :
  - Less then - ‘ < ‘
  - Less then or Equal to - ‘ <= ‘
  - Greater then - ‘ > ‘
  - Greater then or Equal to - ‘ >= ‘
- For ALL relational operations the operands must be of the same type.
- The result for ALL relational operations will be a Boolean value.

© All rights reserved to Abramov B.

Logical Operators

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

- Same as with relational operations logical operation may only be applied to operators of the same type.
- The result will be of the same type as the operators.

© All rights reserved to Abramov B.

Value Assignment

The assignment sign in VHDL is marked :

<=

Example of an assignment to a boolean identifier :

```
break_time <= false;
car_drive  <= motor_on and tank_fueled;
```

© All rights reserved to Abramov B.

Bit

Since VHDL is used to model digital systems, it is useful to have a data type to represent bit values. The predefined enumeration type **bit** serves this purpose.

It is defined as

type bit **is** (‘0’,’1’);

The logical operators for **Boolean** values can also be applied to values of type **bit**, and they produce result of **type bit**.

The operation ‘0’ **and** ‘1’ will produce the result ‘0’.

‘0’ = ‘0’ and ‘1’

The operation ‘1’ **or** ‘0’ will produce the result ‘1’.

‘1’ = ‘1’ or ‘0’

***The operands must still be of the same type*.**

© All rights reserved to Abramov B.

Bit

The difference between types **Boolean** and **bit** is that boolean values are used to model abstract conditions, whereas **bit** value is used to model ***hardware logic level*.**

Thus, ‘0’ represents a **low logic level** and ‘1’ represent a **high logic level**.

Note: Logical values for an object of the type **bit** **must be** written in quotes.

Example of assignment to an identifier of type bit :

```
my_bit <= '1';
```

© All rights reserved to Abramov B.

Bit Vector

The **bit_vector** type is predefines as standard one-dimensional array type with each element being of type **bit**.

The **bit_vector** type is an unconstrained vector of elements of the **bit** type.

The size of a particular vector is specified during its declaration.

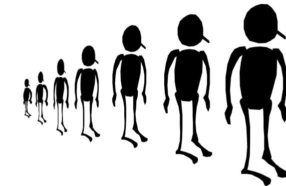
The way the vector elements are indexed depends on the defined range, and can be either ascending or descending .

© All rights reserved to Abramov B.

Bit Vector

left_vector : bit_vector(7 downto 0);

The above line defines a bus of 8 bit width, where the MSB is indexed by the number 7. The LSB is indexed by the number 0. This is a descending range.



right_vector : bit_vector(0 to 7);

This second line defines a bus of 8 bit width, where the LSB is indexed by the number 7. The MSB is indexed by the number 0. This is an ascending range.

The following is an example of an 8 bit register indexed 0 – 7. Where for each index value there is a corresponding logical level value.

0	1	2	3	4	5	6	7
1	0	1	1	1	1	0	0
Register							

© All rights reserved to Abramov B.

Bit Vector

7	6	5	4	3	2	1	0
1	**0**	**1**	**1**	**1**	**1**	**0**	**0**
msb			register				lsb

If bit number 7 is the msb, then it is declared in VHDL as

bit_vector(7 **downto** 0)

and the decimal value of the above register is ***188***.

0	1	2	3	4	5	6	7
0	**0**	**1**	**1**	**1**	**1**	**0**	**1**
msb			register				lsb

If bit number 0 is the msb, then it is declared in VHDL as

bit_vector(0 **to** 7)

and the decimal value of the register is ***61***.

Note : In the above examples the bit numbering was kept the same.

© All rights reserved to Abramov B.

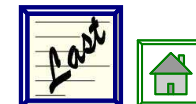
Bit Vector

Assignment to an object of **bit_vector** type can be performed using single element of array (by index), concatenation, aggregates, slices or any combination of them.

Logical value for objects of **bit_vector** type must be written in double quotes.

```
signal example_vector : bit_vector(3 downto 0);
………………………………………………………
example_vector              <= “1001”;
example_vector(2)           <= ‘1’;
example_vector(1 downto 0)  <= “10”;
example_vector(3 downto 1)  <= (‘1’, ‘0’, ‘0’);
example_vector(6 downto 5)  <= (‘0’ & ‘1’);
example_vector              <= (others => ‘1’);
```

© All rights reserved to Abramov B.



Integer

In VHDL, **integer** type is used for whole numbers.

An example of an integer type is the predefined type integer, which includes all the whole numbers in the following range :

integer type identifiers range value is:
(- (2**31) + 1) to ((2 **31) –1).

© All rights reserved to Abramov B.

Integer

Each **integer** in VHDL is represented by a synthesized like bus.

```
my_int1 : integer range 15 downto 0;  -- declared range 4 bit;
my_int2 : integer range 10 downto 0;  -- declared range 4 bit !!!
```

When **integer range** is not declared, then by ***default the range is 32 bit*.**

Example : Declaration of a port of mode in and type integer;

```
port(
      my_int : in  integer ;   -- range 32 bit
       );
```

© All rights reserved to Abramov B.

Integer

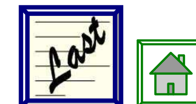
An integer type can be defined either in ascending or descending range.

my_int1 **: integer range** 15 **downto** 0; -- descending range 4 bit;
my_int2 : **integer range** 0 **to** 15; -- ascending range 4 bit

The operations that can be performed on values of **integer** types include the familiar arithmetic operations.

It is an error to assign to an integer object a value which is from outside its range!

© All rights reserved to Abramov B.



Integer

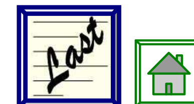
Subtypes **natural** and **positive** are predefined subtypes of **integer**.

natural >= 0;
positive >= 1;

subtype natural is integer range 0 **to integer'high**;

subtype positive is integer range 1 **to integer'high**;

© All rights reserved to Abramov B.



Integer

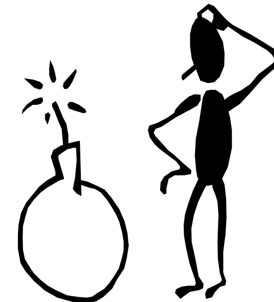
Use Integers with Care !!!

- Synthesis tools create a 32-bit wide resources for unconstrained integers

```
signal Y_int, A_int, B_int : integer ;

Y_int <= A_int + B_int ;
```

- Do not use unconstrained integers for synthesis

```
signal A_int, B_int: integer range -8 to 7;
signal Y_int       : integer range -16 to 15 ;

Y_int <= A_int + B_int ;
```



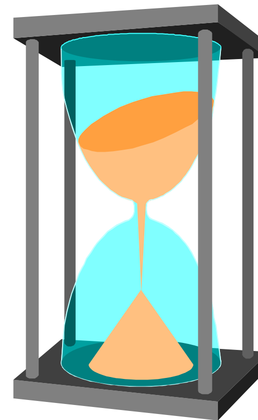
Recommendation: Use integers only as constants or literals

© All rights reserved to Abramov B.

Time

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;
```



NOTE - The time type is not supported by synthesis tools !!!

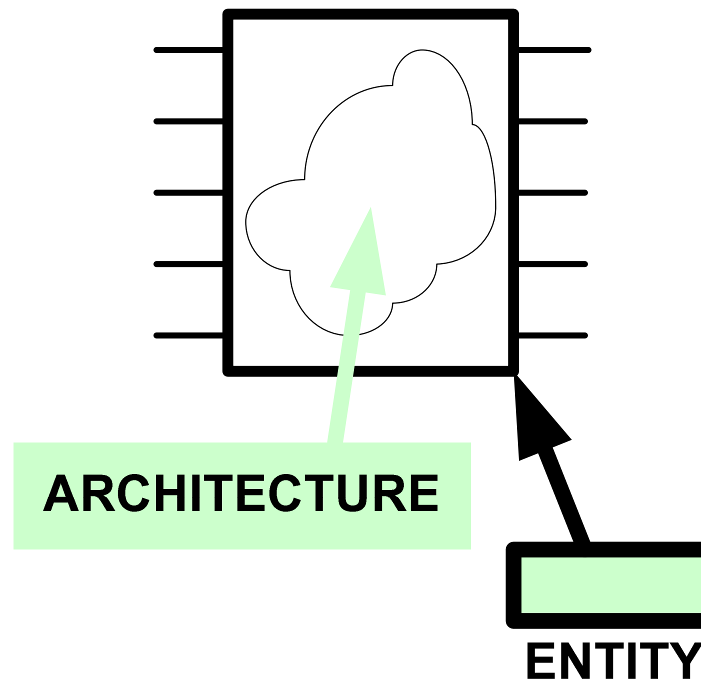
It is used in test benches and modeling.

© All rights reserved to Abramov B.

Architecture

Architecture is the description of logic behavior in order to implement the required design/function.

When looking at an IC the entity is the package with the pins that interface the outer world and the architecture is the implementation of the logic in the IC.



© All rights reserved to Abramov B.

Architecture

Syntax:

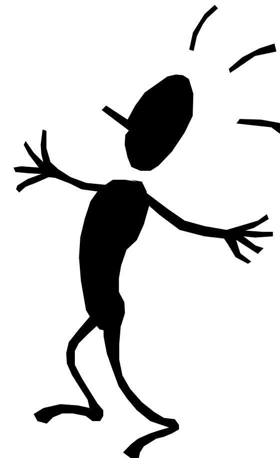
```
architecture arc_entity_name of entity_name is
    -- architecture declaration area
begin
    -- concurrent statements
end architecture arc_entity_name;
```

© All rights reserved to Abramov B.

Architecture

The declaration part of the architecture may contain none, all or any combination of the following language constructs and directives:

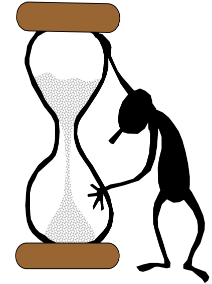
- **Component**
- **Type**
- **Subtype**
- **Constant**
- **Signal**
- **Attribute**
- **Function**
- **Procedure**
- **File**



© All rights reserved to Abramov B.

Architecture

architecture arc_entity_name **of** entity_name **is**



```
architecture arc_entity_name of entity_name is
    component component_name is
                generic(generic_list);
                port (port_list);
    end component component_name ;
    type  some_type;
    subtype subtype_name is base_type range range_constrain;
    constant constant_name : type :=value;
    signal signal_name : type;
    attribute middle : integer;
    attribute middle of signal_name: signal is signal_name'length/2;
    alias my_alias : std_logic_vector(7 downto 0) is my_bus(31 downto 24);

begin
```

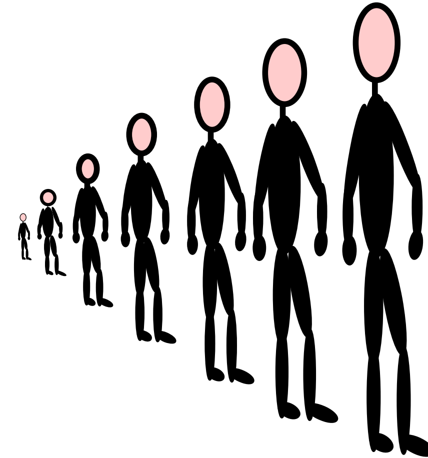
© All rights reserved to Abramov B.

Architecture

- Items declared in the **architecture** are visible in any **process** or **block** within it.
- An **architecture** can contain mix of **component** instances, processes and other concurrent statements.
- An **entity** can have one or more **architectures**.(Which one is used depends on the **configuration)**
- An **architecture** cannot be analyzed unless the **entity** it refers to exists in the same design library.
- The architecture contents starts after the **`begin`** statement. This is called the ***dataflow environment*.**
- All statements in the dataflow environment are executed ***concurrently*** !!!
- Assignment of a value to a port (signal) is done with the assignment - **<=** sign.

© All rights reserved to Abramov B.

Architecture



Example of an even parity calculation :

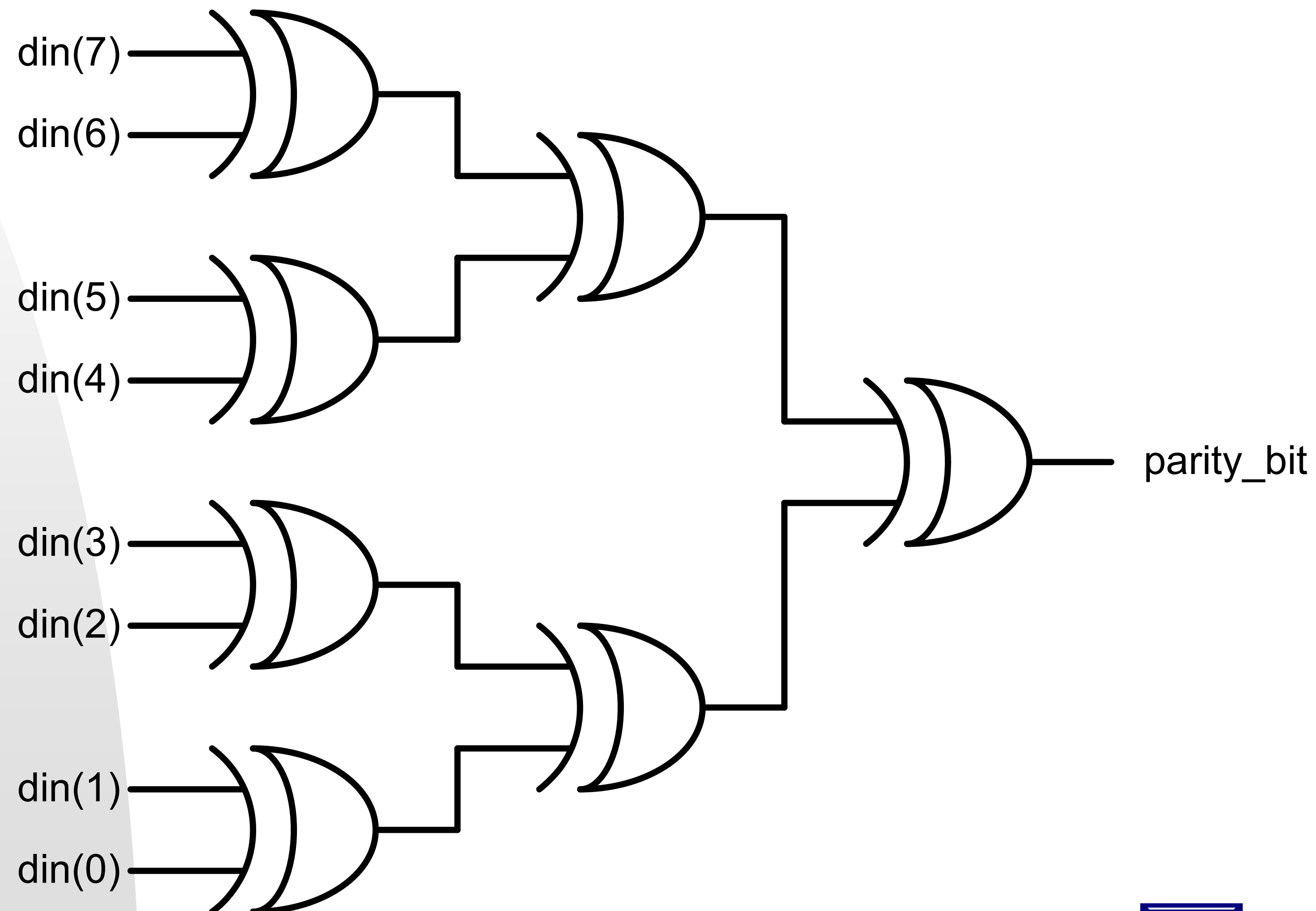
```
architecture arc_parity_check of parity_check is
begin
        parity_bit <= ((din(7) xor din(6))  xor (din(5) xor din(4))) xor
                      ((din(3) xor din(2)) xor (din(1) xor din(0)));
 end arc_ parity_check;
```

Note - A statement can be broken into multiple lines.

© All rights reserved to Abramov B.

Architecture

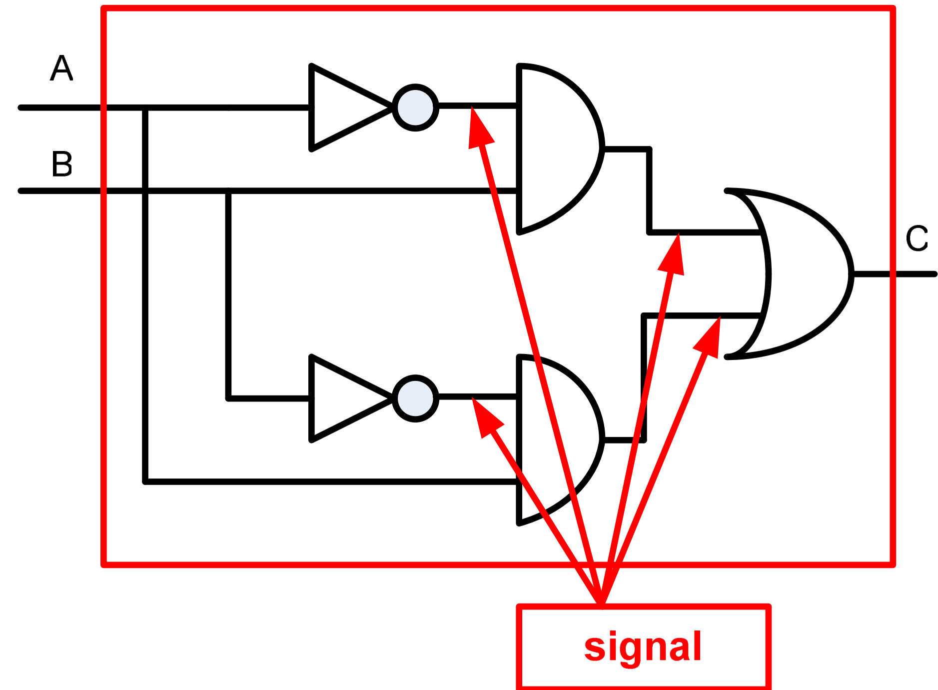
As a result of the above description we will receive the following hardware:



© All rights reserved to Abramov B.

Signal introduction

- A wire transporting data from one element of the design to another, is called **signal**.
- **Not** all wires in a design need to be explicitly declared.



Signals are declared at the architecture before the reserved word – **BEGIN**.

Declaration example:

SIGNAL example_signal : **BIT;**

© All rights reserved to Abramov B.

Component introduction

Structural VHDL:

✓Possibility of creating hierarchical designs built from a set of interconnected ***components.***

Components :

✓Previously coded, simulated, synthesized and placed in design library

Design refinement :

✓Simplification of debugging process

✓The smaller, less complex circuit, the easier debugging process

Component representation:

✓In terms of other, smaller components (structural VHDL)

✓Logic expressions

✓Using behavioural VHDL (mixed style coding)

© All rights reserved to Abramov B.

Component introduction

Structural VHDL:

✓Entity implementation by specifying its composition from subsystems

✓Structural architectural body – system composed only of interconnected subsystems

✓Elements of structural architecture:

1. Signal declaration
2. Component declaration – Specifies the module interface including generics and ports. An exact match of the Entity that the component relates to must be kept.
3. Component instantiation – specifies wire connections between components and between components, ports and logical structures in the current hierarchical level.

© All rights reserved to Abramov B.

Component

Component Example:

Definition : Design a Full adder using two Half adders as building blocks.

Step 1 : Define the functions that will result with proper operation of a Half adder.

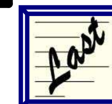
Half adder truth table and function result:

S = in_a xor in_b

C = in_a and in_b

in_a	in_b	c_out	s_out
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

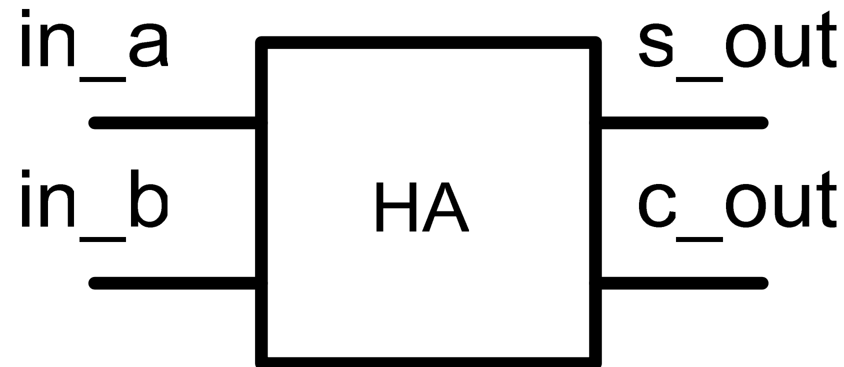
© All rights reserved to Abramov B.



Component

Step 2 : Define the interface of the half adder to the external world and write the entity for it.

```
entity half_adder is
    port (
        in_a  : in   bit;
        in_b  : in   bit;
        s_out : out bit;
        c_out : out bit
        );
end entity half_adder;
```

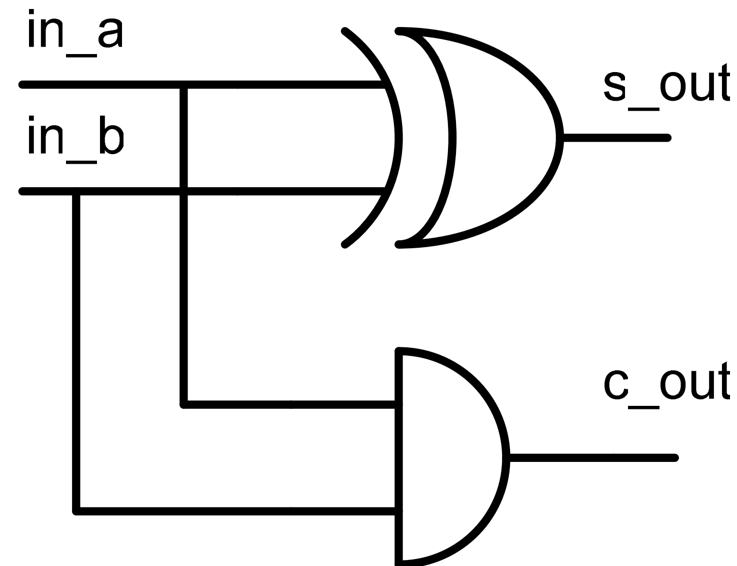


© All rights reserved to Abramov B.

Component

Step 3 : Write the architecture implementation for the resulting functions that implement the design.

```
architecture half_adder_arch of half_adder is

begin

    s_out <= in_a xor in_b;
    c_out <= in_a and in_b;

end architecture half_adder_arch;
```



© All rights reserved to Abramov B.

Component

Step 5 : Define the functions that will result with proper operation of a Full adder.

Full adder truth table and function/s results:

in_a	in_b	in_c	c_out	s_out
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

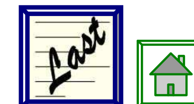
S = (in_b xor in_c) xor in_a

C = (in_b and in_c) or (in_a and (in_b xor in_c))

OR

C = (in_a and in_b) or (in_b and in_c) or (in_a and in_c)

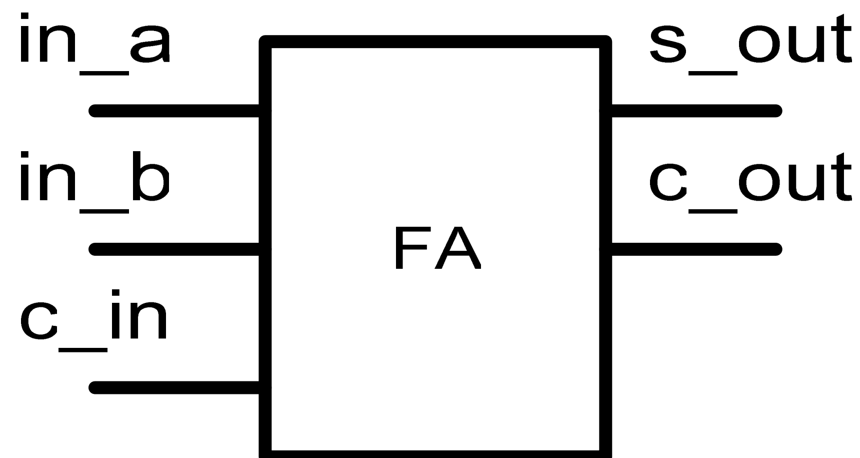
© All rights reserved to Abramov B.



Component

Step 6 : Define the interface of the Full adder and write the entity for it.

```
entity full_adder is
    port (
        in_a  : in   bit;
        in_b  : in   bit;
        in_c  : in   bit;
        s_out : out bit;
        c_out : out bit
        );
end entity full_adder;
```



© All rights reserved to Abramov B.

Component

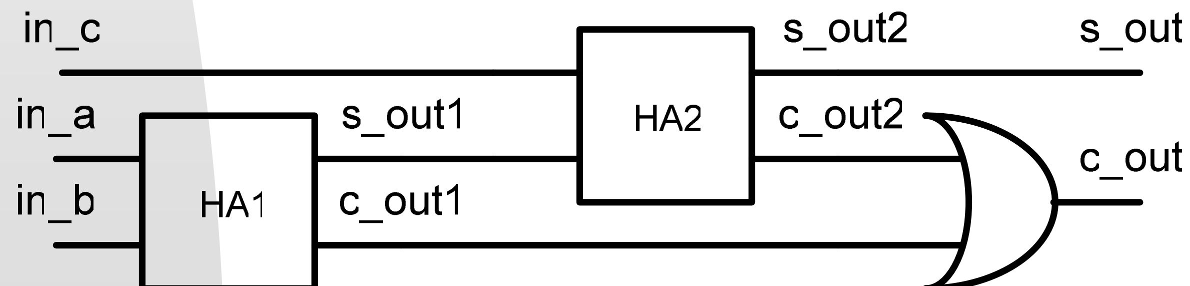
Step 7 : Write the architecture implementation for the design.

```
architecture full_adder_arch of full_adder is

  component half_adder is
      port (  in_a: in    bit;
              in_b: in    bit;
              s_out         : out bit;
              c_out         : out bit );
  end component half_adder;

  signal s_out1, c_out1, c_out2 : bit

begin
```

© All rights reserved to Abramov B.

Component

```
HA1 : half_adder
      -- Instantiation by position
      port map ( in_a, in_b, in_c, s_out1, c_out1);
HA2 : half_adder
      -- Instantiation by name
      port map ( in_a   => in_c,
                 in_b   => s_out1,
                 s_out  => s_out ,
                 c_out  => c_out2);
   c_out <= c_out1 or c_out2;
end architecture full_adder_arch;
```



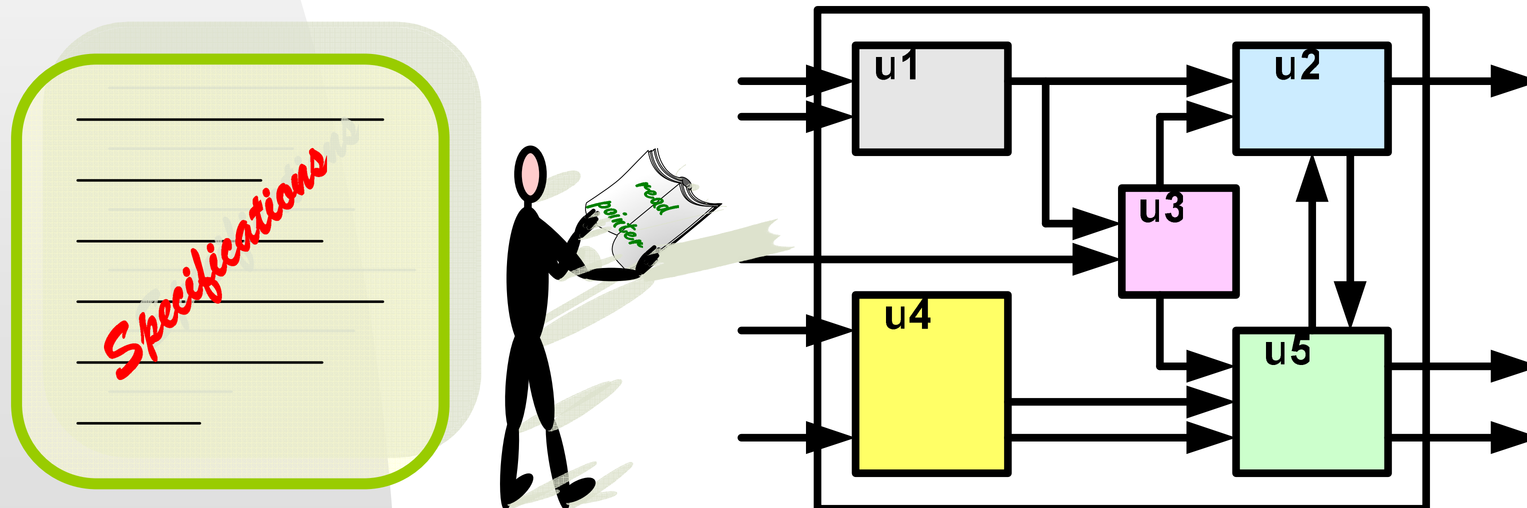
© All rights reserved to Abramov B.

Component

The example just shown is an example of a mixed structural and behavioral architecture modeling.

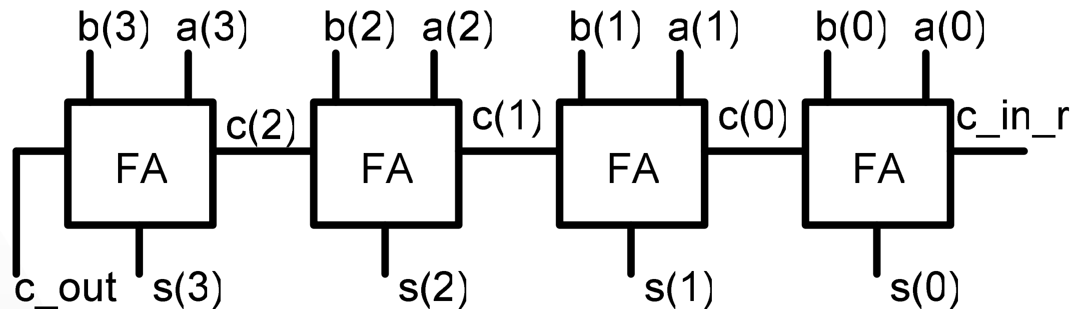
Very often it is easier and more understandable to use a mixed modeling scheme than a pure structural or pure behavioral models.

One place where a pure structural model will most probably be found is at the top most hierarchy of a complex designs.



© All rights reserved to Abramov B.

Component



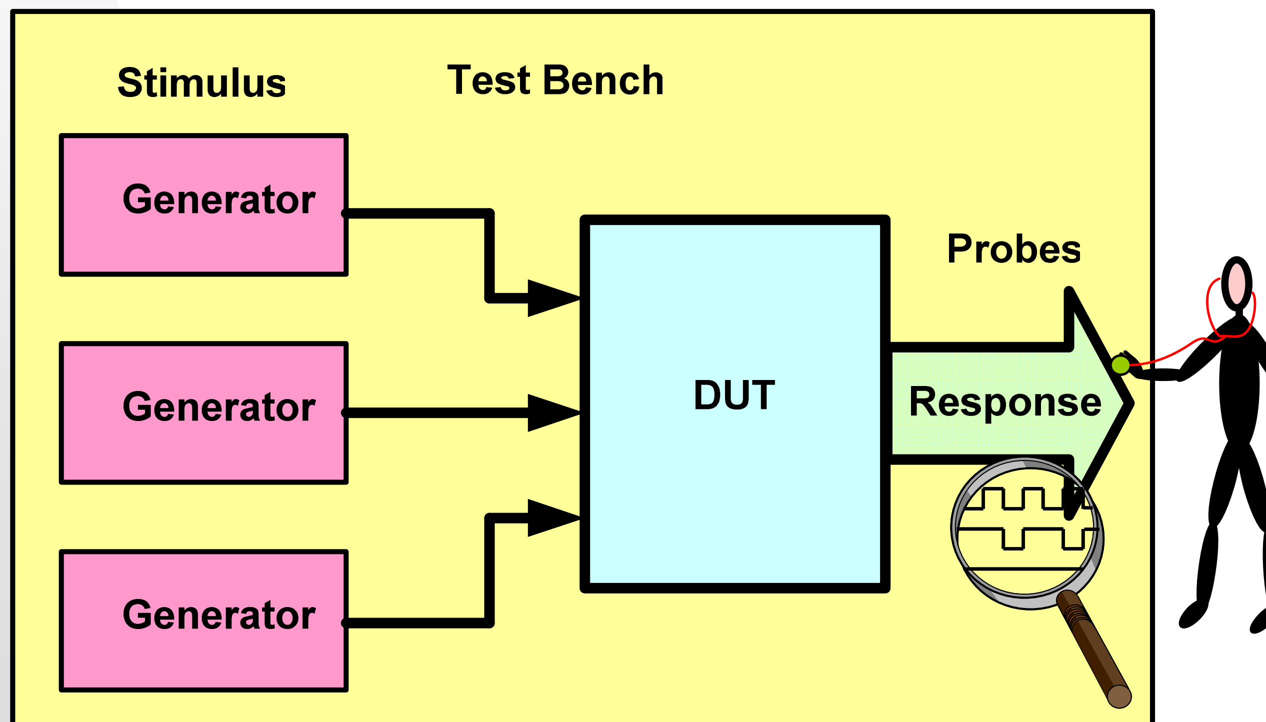
Example for a full structural model can be seen in the described 4 bit ripple adder.

```
architecture arc_ripple_adder4 of ripple_adder4 is
-- component declaration comes here
signal c : bit_vector(2 downto 0);
begin
   fa0: fa
   port map (in_a  => a(0), in_b   => b(0), in_c=>c_in_r,
                  c_out=> c(0), s_out => s(0));
   fa1: fa port map (a(1), b(1), c(0), s(1), c(1));
   fa2 : fa ..................................
```

© All rights reserved to Abramov B.

Test Bench

A design is uncompleted without verification.
The most popular solution is to utilize a ***test benches***.
The name comes from the analogy with a real hardware test bench, on which a device under test is stimulated with signal generator and observed with signal probes.



© All rights reserved to Abramov B.

Test Bench

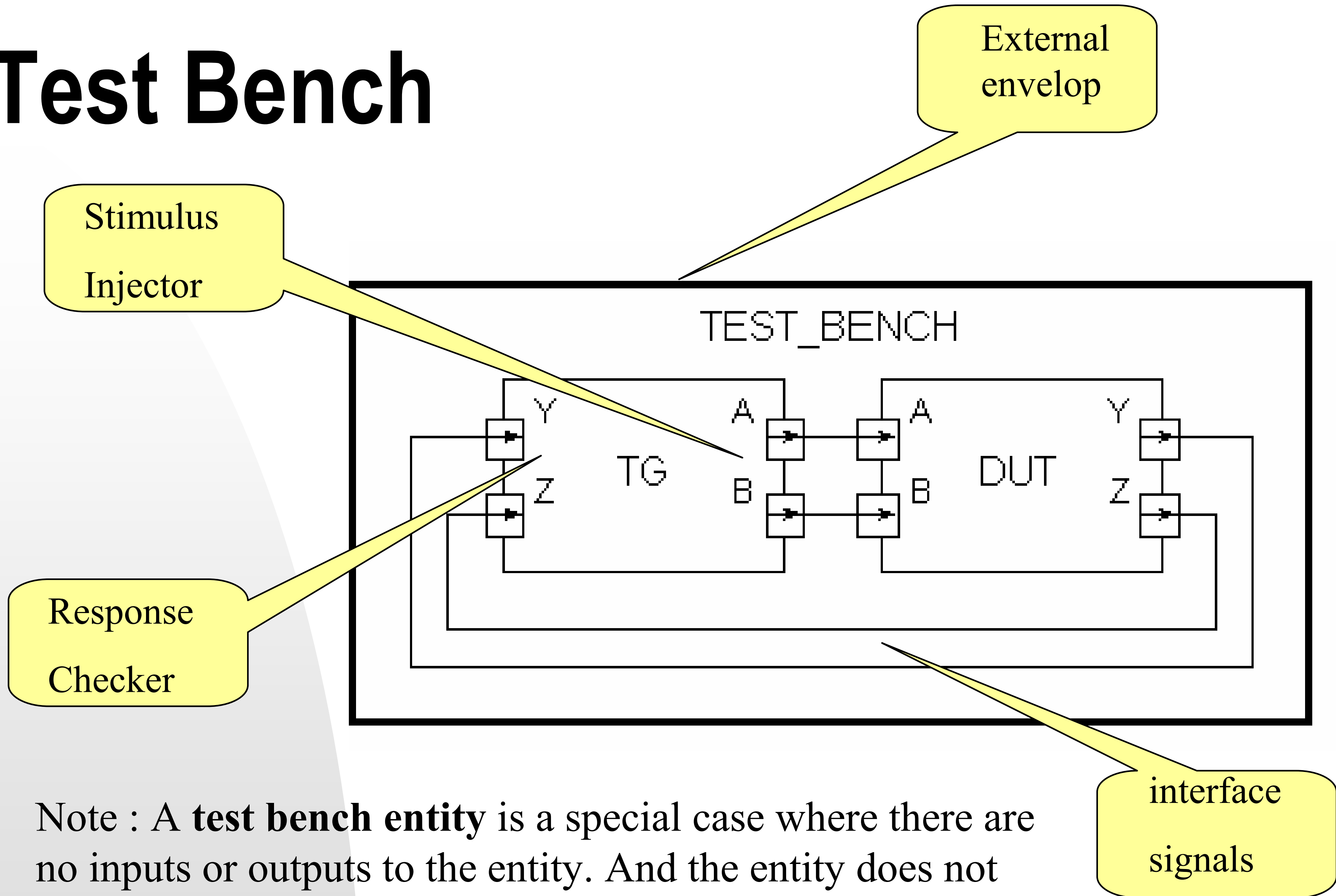
A ***test bench*** is an environment, where a design (often called, design under test – DUT) is checked by applying values (stimulus) to the input ports and monitoring its responses by observing signal probes and monitors at the output ports and internal signals.

A test bench always consists of the following elements:

- A “socket” for the DUT.
- Stimulus generator.
- Tools for monitoring the DUT response to the injected stimulus.

© All rights reserved to Abramov B.

Test Bench



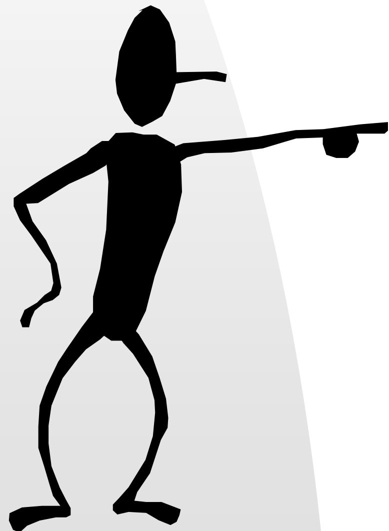
Note : A **test bench entity** is a special case where there are no inputs or outputs to the entity. And the entity does not need to interact with any external units.

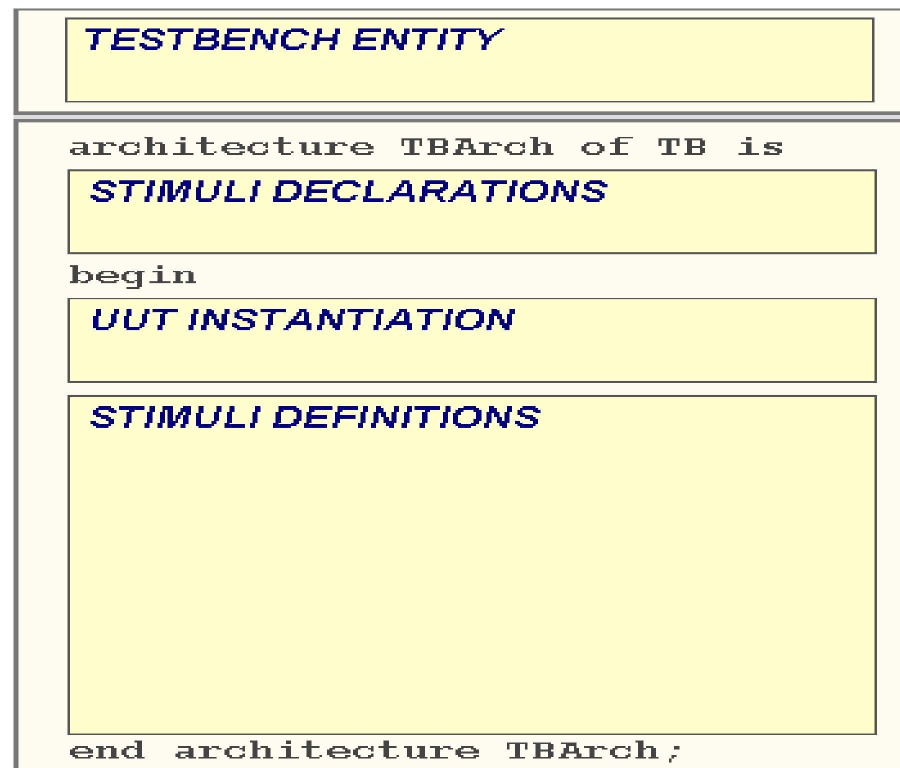
© All rights reserved to Abramov B.

Test Bench

A VHDL *test bench* consists of:

- **an empty entity declaration,**
- **an architecture body containing an instance of the unit to be tested**
- **generators that inject sequences of values to the inputs of the tested unit**
- **receptors to absorb the results from the tested unit**





© All rights reserved to Abramov B.

Simple Test Bench

```
entity ripple_adder4_tb is
end entity ripple_adder4_tb;

architecture arc_ripple_adder4_tb of ripple_adder4_tb is
    component  ripple_adder4 is
            port(
                a       : in    bit_vector(3 downto 0);
                b       : in    bit_vector(3 downto 0);
                c_in    : in    bit;
                s       : out   bit_vector(3 downto 0);
                c_out   : out   bit
                  );
    end component ripple_adder4;
```

© All rights reserved to Abramov B.

Simple Test Bench

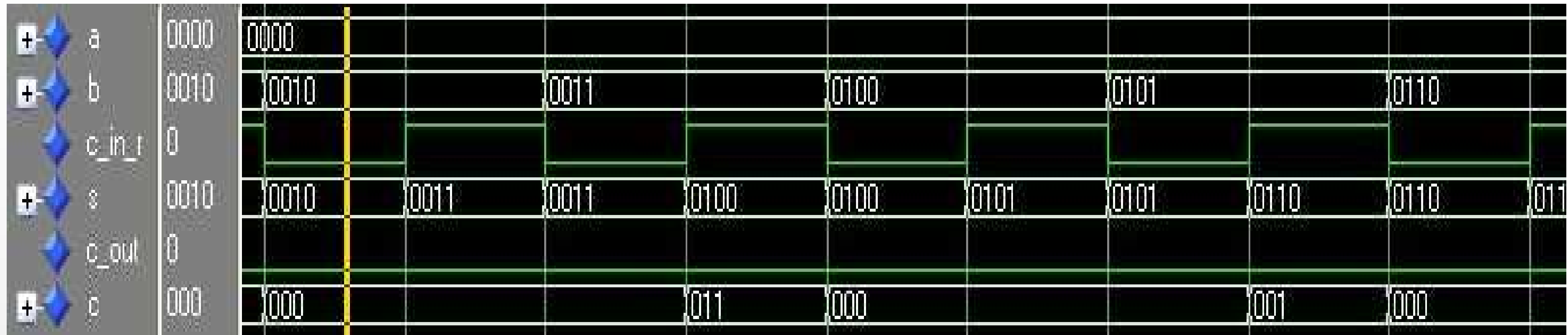
```
signal  a        : in   bit_vector(3 downto 0) := x"0";
signal  b        : in   bit_vector(3 downto 0) := x"0";
signal  c_in     : in   bit : '0';
signal  s        : out  bit_vector(3 downto 0);
signal  c_out    : out  bit;

begin
uut: ripple_adder4
    port map (a, b, c_in, s, c_out);
a <= "0000", "0001" after 1600 ns, "0010" after 3200 ns, "0011" aft…  ;
b <= x"0", x"1" after 100 ns, x"2" after 200 ns, x"3" aft…
c_in <= not c_in after 50 ns;
end architecture arc_reg_tb;
```

Note : Interval time written must be relative to start time and not to previous transaction **!!!**

© All rights reserved to Abramov B.

Simple Test Bench

Wave diagram of the 4 bit ripple adder simulation:



The diagram shows the sum and carry results for the following values:

- a at “0000”
- b at “0010” to “0110”
- c_in toggles every 50 ns from ‘0’ to ‘1’ and vice versa.

© All rights reserved to Abramov B.