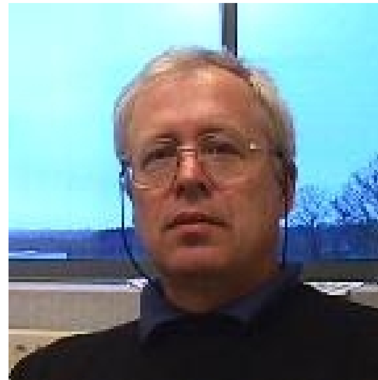
VHLD code reuse

P. Bakowski



bako@ieee.org

VHDL for reuse ?

VHDL – IEEE 1076 — 1987

VHDL – IEEE 1164 — 1991

VHDL – VITAL — 1993

VHDL – IEEE 1076.3 — 1995

VHDL – AMS 1076.6 — 1998

analog extension for mixed-signal modeling

Coding mechanisms for reuse

- basic coding practices
- generics
- generate statements
- unconstrained arrays and attributes
- configurations

Coding recommendations

- develop naming convention for the design,
- use short but meaningful names: data_bus, dec_sel,..
- use upper case for names of labels and constants.
- use underscore for active low signals: reset_low, clock_low, ..
- for vectors and arrays use consistent (standard ordering) ordering: downto or to, not both
- for documentation reasons include a header at the top of every source file, including scripts

VHDL file header

- File name
- Author name and affiliation
- Date the file was created/modified
- Description of the functions and list of key features
- Version number
- Modification history: the list of changes with date and author

VHDL file header

```
-- File:  simprisc.vhd
-- Author: Jean-Luc  Dubois, ARMEL
-- 21/07/03
 -- simple risc processor
 -- the processor works in three
 -- stages: fetch, decode, execute
   -- version 1.1
   -- modification history
     -- 01/02/04  JLD  original version 0.1
-- 11/02/04  JLD  mply instruction
-- added   version  1.0
-- 21/07/04  PB   mply optimised
-- version  1.1
```

VHDL file header

For entities declare ports in a logical order; declare one port per line with a comment in the same line.

```
use work.mestypes.all;
entity risc is
port (clk: in std_logic, -- clock signal
      rst: in std_logic, -- reset signal
      dbus: inout dataword, -- in package mestypes
      abus: out adressword,
    rd: out std_logic,        -- read memory
    wr: out std_logic         -- write memory );
end entity;
```

Port map associations

For port map use named association rather than positional association.

```
U_add: mon_add
generic map (delai => 10 ns)
port map(a => in1,b=>in2,cin => in3,
sum => out1, cout => out2);
```

Flexible functions

If possible the functions should be flexible and reusable in different designs.

```
-- converts a bit to std_logic
function Bit2Std(valbit: bit) return
std_logic is
begin
  case valbit is
    when '0' => return '0';
    when '1' => return '1';
  end case;
end Bit2Std;
```

Flexible functions

```
function Bit2Std(vecteur: bit_vector) return
std_logic_vector is
alias vecteur_n: bit_vector(vecteur'length -
1 downto 0) is vecteur;
variable var:
std_logic_vector(vecteur_n'range);
begin
for i in vecteur_n'range loop
var(i) = Bit2Std(vecteur_n(i));
-- call to above function (overloading)
end loop;
return (var);
end Bit2Std;
```

Loops and arrays

Use arrays instead of loops. The arrays are significantly faster to simulate than loops.

```
function my_xor_loop(bbit: std_logic; avec:
std_logic_vector(x downto y)) return
std_logic_vector is
variable var: std_logic_vector(avec'legth-1
downto 0);
    begin
       for i in var'range loop
          var(i) = avec (i) xor bbit;
              end loop;
          return (var);
            end my_xor_loop;
```

Loops and arrays

```
function my_xor_array(bbit: std_logic; avec:
std_logic_vector(x downto y))
return std_logic_vector is
variable var, temp:
std_logic_vector(avec'legth-1 downto 0);
begin
temp := (others => bbit);
      var = temp xor bbit;
return (var);
end my_xor_array;
```

Libraries and portability

Use IEEE standard types and the subtypes based these IEEE types. The standard IEEE library includes logic and arithmetic types:

```
IEEE.std_logic
IEEE.numeric_bit
IEEE.numeric_std
```

Libraries and portability

```
IEEE.std_logic

IEEE.numeric_bit

IEEE.numeric_std
```

The IEEE types may be used to create new subtypes:

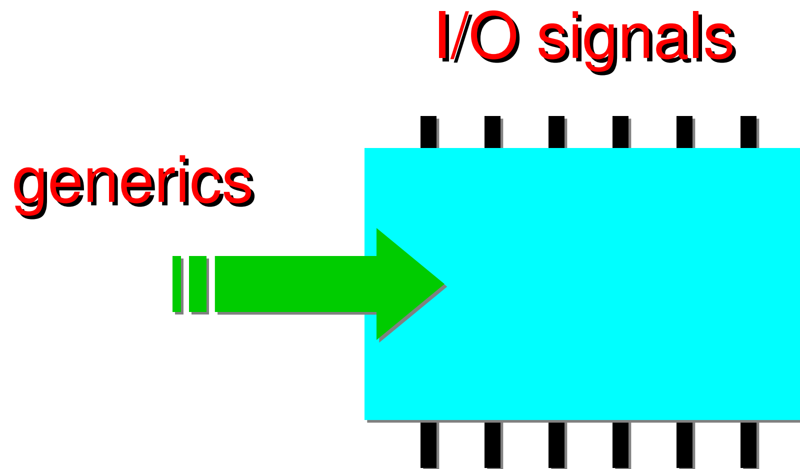
```
subtype MOT is std_logic_vector(31 downto 0);

subtype NUMMOT is unsigned(31 downto 0);
```

Do not create too many subtypes.

Generics

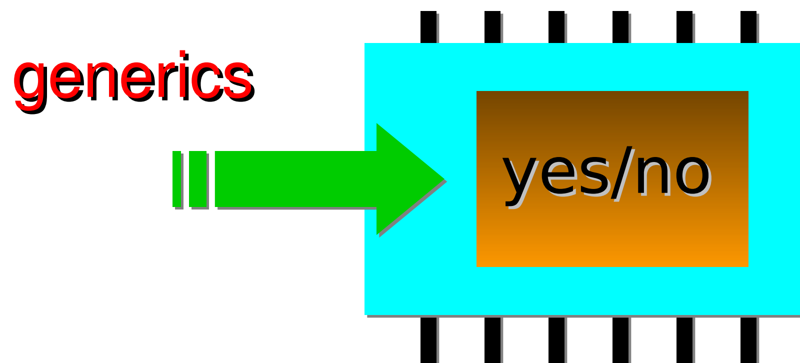
The generics allow us to modify/adapt both the structure and the behavior of the circuit.



Generics

The generics allow us to modify/adapt both the structure and the behavior of the circuit.

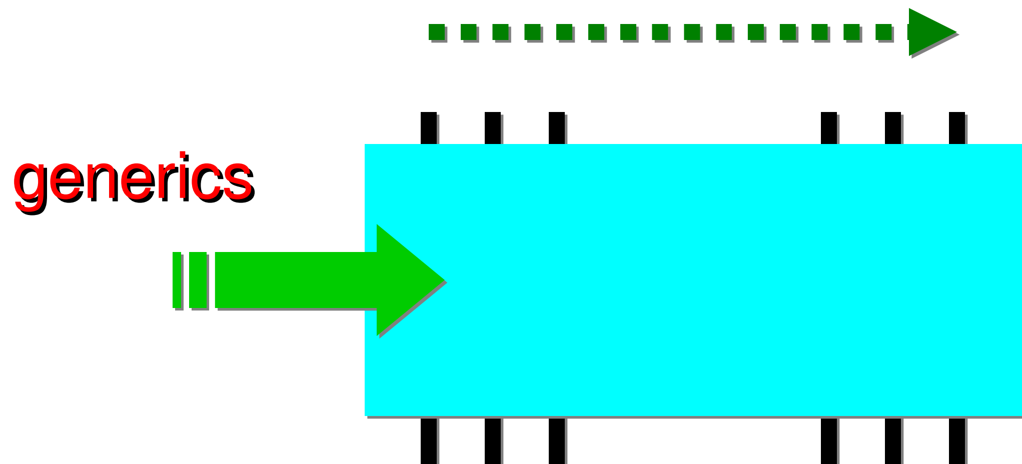
- structure through including/excluding some part of the circuit



Generics

The generics allow us to modify/adapt both the structure and the behavior of the circuit.

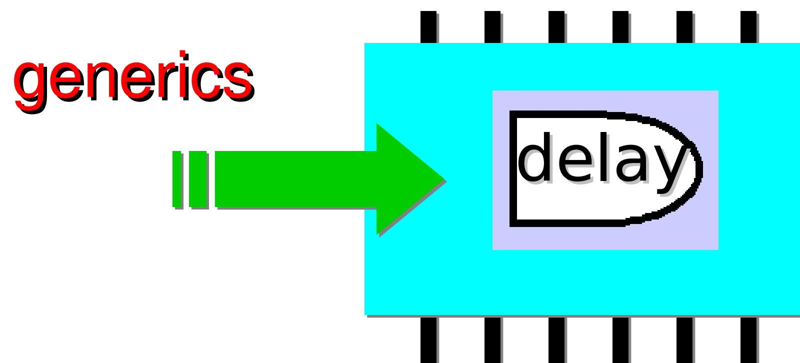
- structure through widening/narrowing the data paths



Generics

The generics allow us to modify/adapt both the structure and the behavior of the circuit.

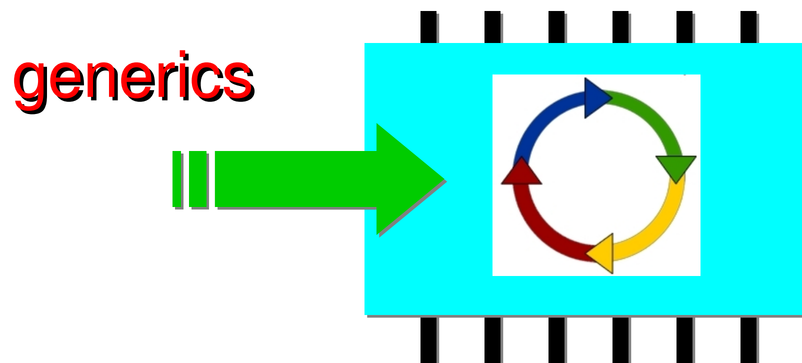
- behavior through modifying the timing characteristics



Generics

The generics allow us to modify/adapt both the structure and the behavior of the circuit.

- behavior through modifying the number of functional steps



Generics for structure/behavior

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity compteur is
generics(bit_width:integer:=3;
-- structure : size of the bus
cmpt_enable:integer:=1;
-- structure : selection of some circuit
down_cmpt:integer:=0;
-- behavior: number of steps
delai:time:= 3 ns; -- behavior : delay );
       port(clk,reset,en:in std_logic;
compt:out unsigned(bit_width-1 downto 0));
end compteur;
```

Generics for structure/behavior

```
architecture  behaviour  of compteur  is
signal compt_s: unsigned(bit_width-1 downto 0);
begin
process(clk,reset)
 begin
if(reset ='0') then  compt_s <= (others=>'0');
   elsif rising_edge(clk) then
if(cmpt_enable =1) then -- structure: removed if false
   if(en='1') then
if(down_cmpt=0) then  compt_s <= compt_s+1;  -- function
else compt_s <=compt_s-1;
end if; end if;
else
if(down_cmpt=0) then compt_s <= compt_s+1;   -- function
 else compt_s <=compt_s-1;
 end if; end if;
 end process ;
 compt <= compt_s after delai ; -- function
end behaviour;
```

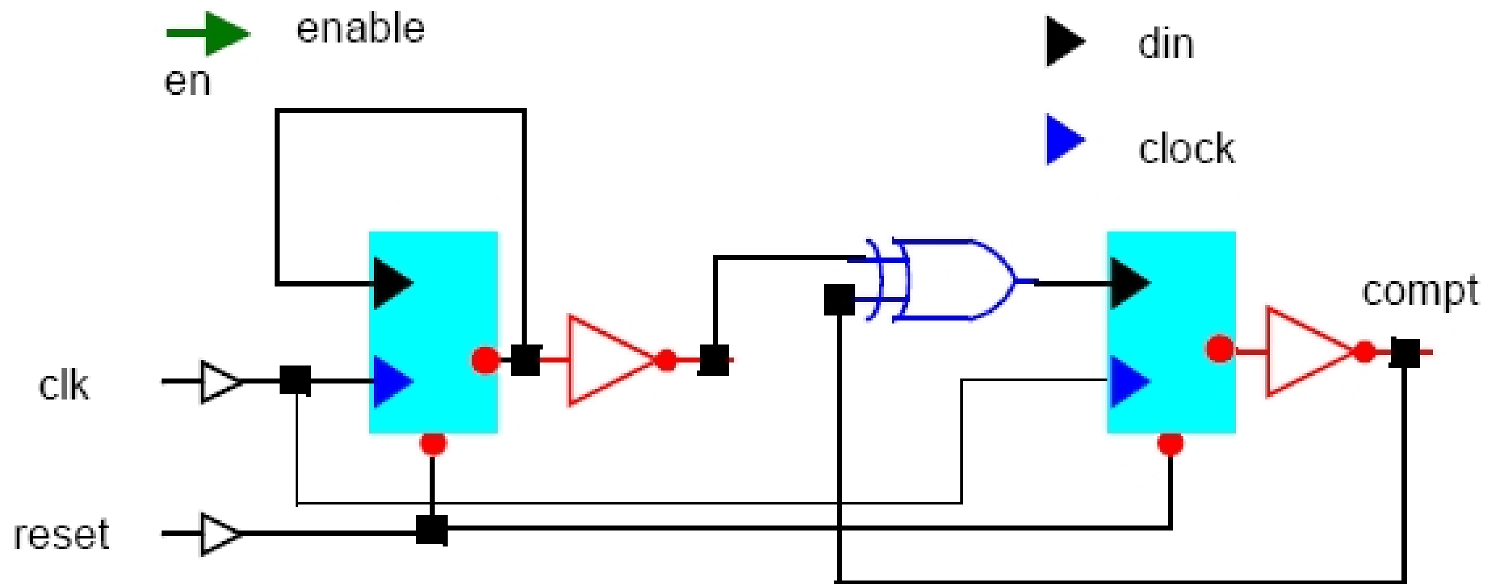
Generics for structure/behavior

Let us consider a set of generic values for this counter.

```
bit_width=> 2     -- two bit counter
cmpt_enable=> 0 -- enable circuit not valid
down_cmpt=> 0     -- up counter validated
delai=> 5 ns      -- no meaning for synthesis
```

Generics for structure/behavior

```
if(cmpt_enable =1) then
          -- structure: removed if false
```



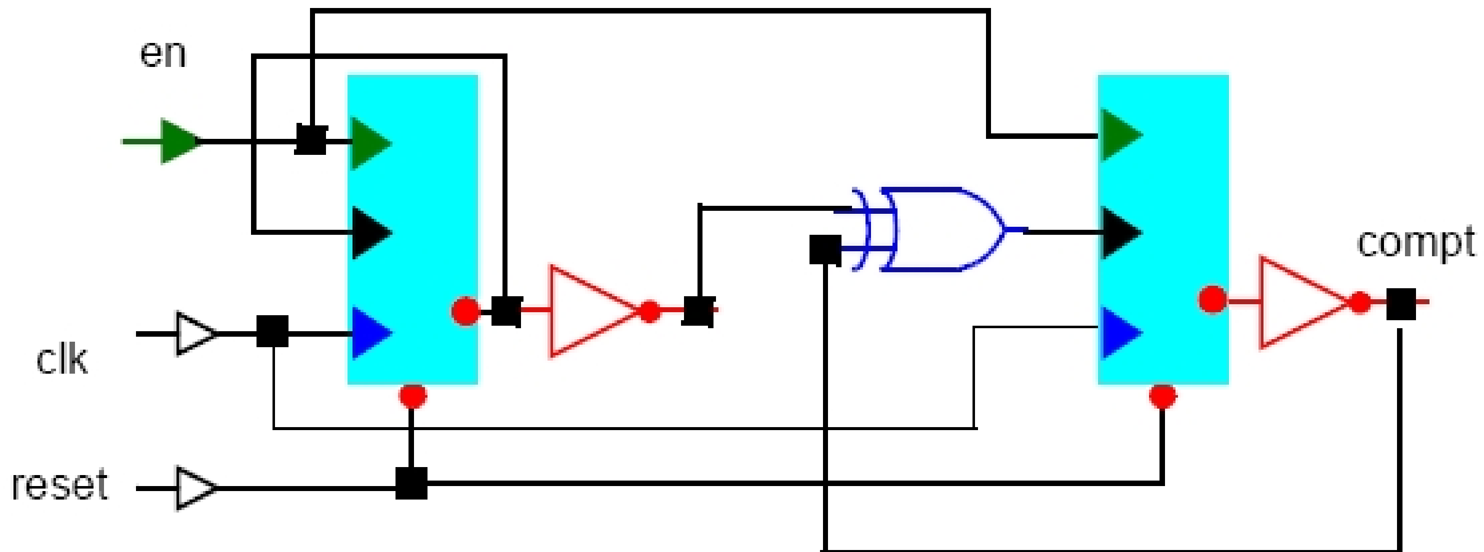
Generics for structure/behavior

Let us consider a set of generic values for this counter.

```
bit_width=> 2    -- two bit counter
cmpt_enable=> 1 -- enable circuit validated
down_cmpt=> 0    -- up counter validated
delai=> 5 ns     -- no meaning for synthesis
```

Generics for structure/behavior

```
if(cmpt_enable =1) then
        -- structure validated
```



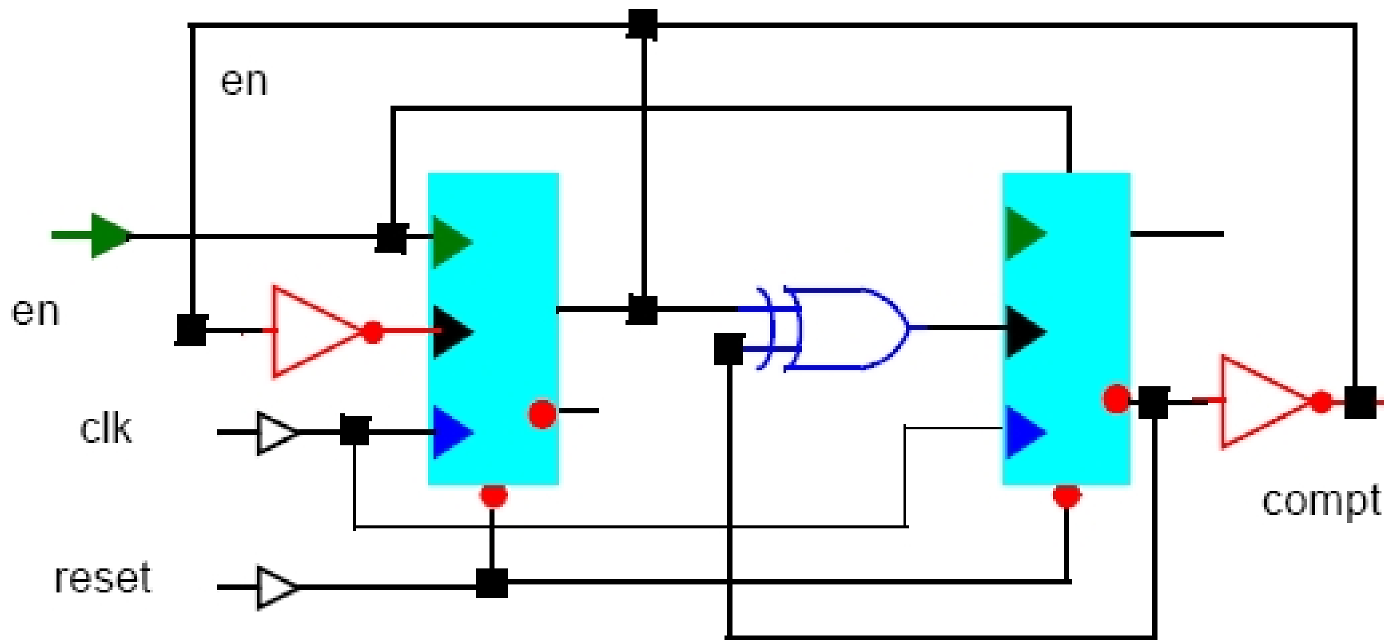
Generics for structure/behavior

Let us consider a set of generic values for this counter.

```
bit_width=> 2     -- two bit counter
cmpt_enable=> 1 -- enable circuit validated
down_cmpt=> 1     -- down counter validated
delai=> 5 ns      -- no meaning for synthesis
```

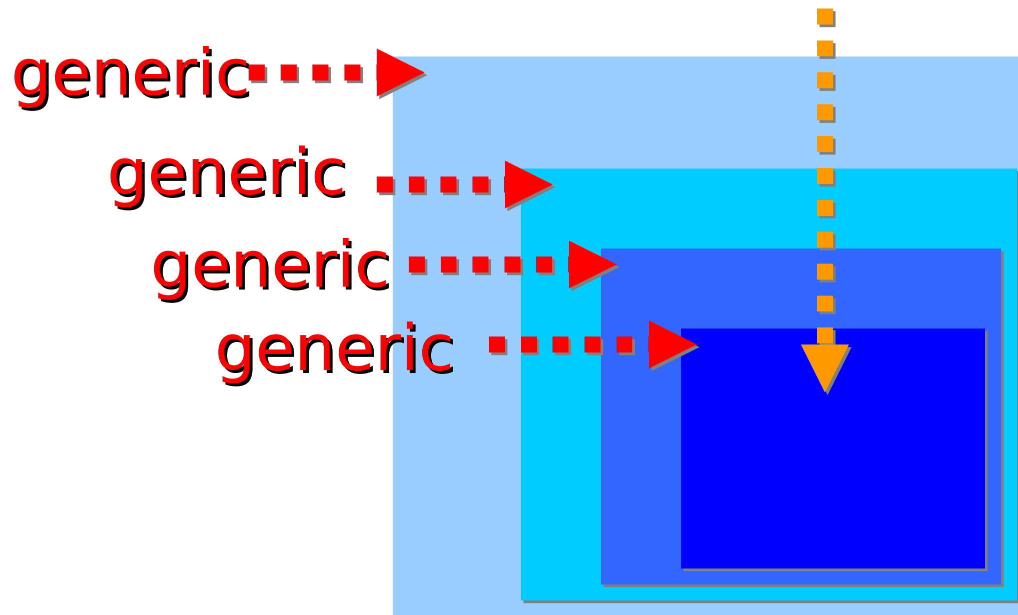
Generics for structure/behavior

```
if(down_cmpt=0) then  compt_s <=compt_s+1;
                else compt_s <=compt_s-1;
```



Constants and packages

One problem with generics is that you use them in a hierarchy; their values must pass down to the lowest level through whole hierarchy.



Constants and packages

An efficient way to avoid these problems is the use of packages and constants.

```
package pack is
constant ..

end pack;
```

Constants and packages

The following entity has no generic parameters in it, instead we have introduced a number of constants defined in the package `par_pack.vhd`.

This package can be compiled separately and used in several design units: `entity, architecture`

Constants and packages

```
package par_pack is
  constant bit_width: integer:=8;
subtype cmpt_width is integer
range bit_width-1 downto 0;
constant cmpt_enable: integer:=1;
constant down_cmpt: integer:=0;
constant delai: time:= 5 ns;
end par_pack;
```

Constants and packages

```
library ieee;
 use ieee.std_logic_1164.all;
  use ieee.numeric_std.all;
   use work.par_pack.all;

entity compteur is
port(clk,reset,en:in std_logic;
cmpt:out unsigned(cmpt_width));
end compteur;
```

Deffered constants

Deferred constants are declared but not initialized in a package.

They are initialized directly in the design units.

In any case the deferred constants must be defined before use.

Deffered constants

```
package par_pack_deff  is
constant bit_width: integer:=8;
  -- cannot be deffered
subtype cmpt_width is integer range bit_width-1
downto 0;
constant cmpt_enable: integer;  -- deffered

constant down_cmpt: integer;    -- deffered

constant delai: time:= 5 ns;
end par_pack_deff;
```

Deffered binding

```
use work.par_pack_deff.all;
entity compteur is
port(clk,reset,en:in std_logic;cmpt:out
unsigned(cmpt_width));
end compteur;

architecture behaviour of compteur is
   -- instantiated deffered binding
constant cmpt_enable: integer:=1;
constant down_cmpt: integer:=0;
 signal cmpt_s:unsigned(cmpt_width);
    begin
          ...
    end behaviour ;
```

Generate statements

The `generate` statements allow us to instantiate regular structures very efficiently.

Combined with generics, `generate` statements offer a very powerful mechanism to build (generate) complex but regular systems.

The following example shows the structural instantiation of a register from flip-flop components.

Generate statements

```
entity regn is
generic(n : positive:=4);
port( clock,reset,wen:in std_logic;
      din:in std_logic_vector(n-1 downto 0);
qout:out std_logic_vector(n-1 downto 0));
end regn;
```

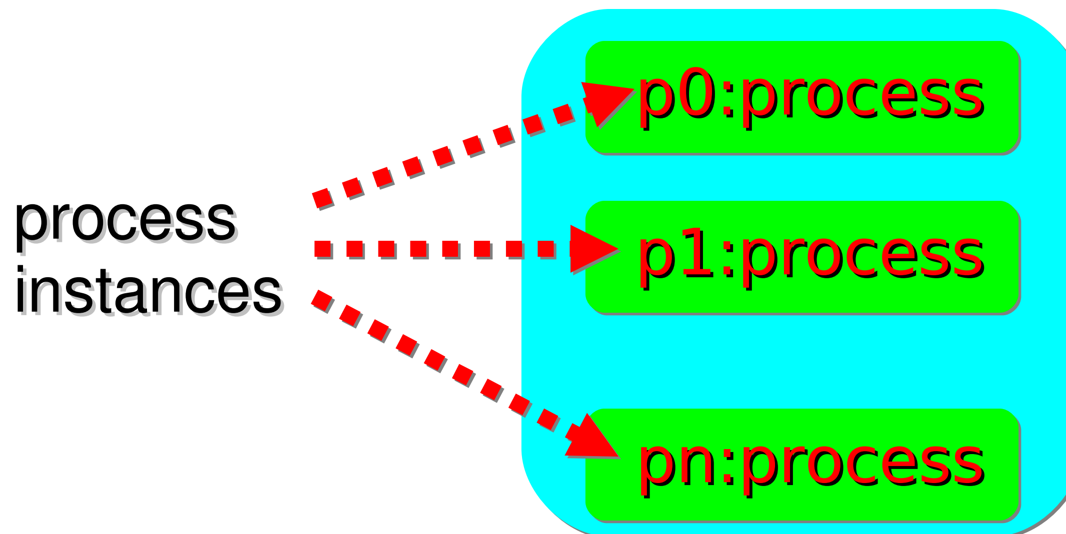
Generate statements

```
architecture structure of regn is
component ff
port(rst,clk,we,d,q);
end component;
begin
for i in 0 to n-1 generate
unit:ff
port map(clock,reset,wen,din(i),qout(i));
    -- n units instantiated:
   -- unit0, .. , unit(n-1)
 end generate;
end structure;
```

Generate statements

The following example also generates a model of register, this time multiplying the processes (bff0, bff1, .. bffn).

```
for i in 0 to n generate ..
```




```
end generate ..
```

Generate statements

This approach may be considered as a generation of behavior.

```
entity regn is
generic(n : positive:=4);
port(clock,reset,we:in std_logic ;
din:in std_logic_vector(n-1 downto 0);
qout:out std_logic_vector(n-1 downto 0));
end regn;
```

Generate statements

```
architecture behaviour of regn is
begin
   for i in 0 to n-1 generate
     -- generation of n processes
bff:process (reset,clock)
 begin
if(reset ='0') then qout<='0';
   elsif (clock'event and clock ='1')
then
if(we='1') then qout(i) <=din(i);
end if;end if;
end process ;
end generate ;
end behaviour;
```

Conditional generate statements

The use of conditional generate statements allows to enable or disable logic that implements certain elements instead of manually removing the code.

The following is an example of conditional code selection.

The output of circuit may be assigned synchronously or as synchronously depending on the value (state) of the sync_out parameter.

Conditional generate statements

```
sync:if(sync_ou ='1') generate
process (clk)
begin
if(clock'event and clock ='1') then
if(rd='1') then qout <= din; end if;
end if;
end process;
end generate ;
async:if( sync_ou='0') generate
process (rd)
begin
if(rd='1') then qout <= din; end if;
end process ;
end generate ;
```

Unconstrained arrays

Unconstrained arrays are necessary to build flexible and reusable models.

VHDL allows the designer to use unconstrained arrays without specifying the index bounds.

Instead of fixed bounds we specify the corresponding attributes (`'range`, `'high`, `'low`).

Unconstrained arrays

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
   -- includes + operator
entity compunconstr is
port(clock,reset:in std_logic;
     count:in unsigned) -- unconstrained
  end compunconstr;
```

will be defined by instantiation: port map (..)

Unconstrained arrays

```
architecture behaviour of counter is
signal s_count:unsigned(count'range);
begin
process (clock,reset )
egin
if(reset ='0') then
s_count<=(count'range=>'0');
elsif (clock'event and clock='1')
then s_count <= s_count+1;
end if;
end process ;
end behaviour ;
```

Unconstrained arrays

The reusable functions and procedures should use unconstrained arrays; their arguments need to be independent from bit widths.

Let us consider a binary to gray code converter.

Unconstrained arrays

```
function bintogray(x:std_logic_vector)
return std_logic_vector is
   variable y:std_logic_vector(x'range);
begin
for i in x'range loop
if (i=x'left) then y(i) := x(i);
else y(i) := x(i) xor x(i+1);
end if; end loop;
return y;
end;
```

Unconstrained arrays

Most of the standard functions prepared in the IEEE packages use unconstrained arrays.

to_integer() and to_unsigned()

Unconstrained arrays

```
library ieee;
use ieee.std_logic_1164.all;
-- standard logic types and operators
use ieee.numeric_std.all;
-- standard numeric types and operators
integer intval;
unsigned mon_vect(0 to 7);
intval = to_integer(mon_vect);
mon_vect = to_unsigned(intval,8);
```

unsigned

Configurations

The configuration statements allow us to bind the selected components to the instances.

The configuration may be used with generics to select an architecture for entity or to override port mappings in an instantiation.

Configurations

```
architecture buffered of compteur is
component tampon
port(entree:in std_logic;
    sortie:out std_logic);
end component;
signal cmpt_s:unsigned(cmpt_width-1 downto 0);
begin
process (clk,reset)  begin ...
end process;
genbuff:for i in bit_width-1 downto 0
generate
tamp:tampon port map(cmpt_s(i), cmpt(i));
end generate;
end buffered;
```

Configurations

Now we have two architectures possible for the counter entity.

The following model represents a top unit containing two counters:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity twocounters is
port (..)
end twocounters;
```

Configurations

```
architecture structural of twocounters is
constant bit_width: integer:=4;
signal clock,enable,reset: std_logic;
dcount,ucount:unsigned(bit_width-1 downto 0);

component compteur
port(clk,reset,en:in std_logic;
cmpt:out unsigned(cmpt_width-1 downto 0));
end component;

begin

..
end structural;
```

one component declaration : to be taken with one or more configurations

Configurations

```
architecture structural of twocounters is
constant bit_width: integer:=4;
signal clock,enable,reset: std_logic;
dcount,ucount:unsigned(bit_width-1 downto 0);
component compteur
port(clk,reset,en:in std_logic;cmpt:out
unsigned(cmpt_width-1 downto 0));
end component;
begin
iupcompt: compteur  -- first instance
port map(clock,reset,enable,ucount);
idocompt: compteur  -- second instance
port map(clock,reset,enable,dcount);
end structural;
```

Configurations

```
configuration conf of twocounters is
for structural  -- architecture
for iupcompt:  -- component of architecture
compteur use entity work.counter(buffered)
generic map(cmpt_enable=>1, cmpt_down=> 0);
for buffered
for genbuff -- configure generate statement
for tamp: tampon use entity
work.tampon(rtl) end for;
end for;
end for;
end for;
```

Configurations

no internal components

```
for idocompt:
compteur use entity work.counter(behaviour)
generic map(cmpt_enable=>1, cmpt_down=> 1);
end for;
end for;
end conf;
```

Summary

- VHDL for reuse
- flexible-reusable functions
- generics for structure and behavior
- packages and deffered constants – late binding
- generate statements
- unconstrained arrays
- configurations