Verilog & VHDL : two languages for digital system design

J.L. Noullet CHIPYARDS *rev. 2g*

HISTORY

VHDL

Origin :

Project V.H.S.I.C. (Very High Speed Integrated Circuit) of the D.O.D. (Department of Defense, American Army) (1983).

Goal :

Define a formal language for digital systems **specification**.

Means :

A group of programming language experts, already involved in the development of Ada (a general purpose programming language)

Results :

Standard **IEEE 1076B** (1987), L.R.M. (Language Ref. Manual) (1993), package **IEEE 1164** (1993), Standard **IEEE 1076-2002**, Standard **IEEE 1076-2008** (plus the **A**nalog **M**ixed-**S**ignal extension, not addressed here)

Use :

Specification, Simulation, Synthesis

Verilog

Origin :

A need expressed by the logic simulator users, for having a unique tool for structural and behavioral models.

Goal :

Increase the **productivity** of digital systems designers.

Means :

A group of developers in a small company (Gateway Design Automation, Philip Moorby) selling CAD tools.

Results :

Simulator Verilog-XL (1985), Public Domain (1991),
Standard **IEEE 1364-1995**, **IEEE 1364-2001** (plus the **A**nalog **M**ixed-**S**ignal extension, the System-Verilog extension, **IEEE 1800-2009,** not addressed here)

Use :

Specification, Simulation, Synthesis

Common Concepts :

- ***Structural*** representation : interconnected elements (cells, blocks), described in a hierarchical way.
- ***Behavioral*** representation : a functionality described independantly of its future implementation.

for the behavioral representation :

- ***Concurrent*** mode : data propagate in elements the behavior of which is described by declarations which have a permanent effect (like in structural representations)
- ***Procedural*** mode : data are processed by sequences of instructions (like in computer programs)

VHDL

- Everything is contained in ***entities***, which are nodes of hierarchical trees
- Concurrent mode : data carried by ***signals***
- Procedural mode : data carried by ***variables*** or ***signals*** (option)
- Procedural mode : statements embedded in ***processes***

Verilog

- Everything is contained in ***modules***, which are nodes of hierarchical trees
- Concurrent mode : data carried by ***nets***
- Procedural mode : data carried by “registers” (***regs***)
- Procedural mode : statements embedded in ***procedural blocks***

STYLES

VHDL

- Redundancy : repetition of entity interface specification as component declaration in each architecture.
- Support for multiple (optional) architectures in an entity
- logic types to be defined externally (built-in BIT type unusable)
  `ex: IEEE.std_logic_1164`
- strengths and resolutions to be defined externally
  `ex: IEEE.std_logic_1164`
- integer-vector conversion to be defined, no implicit conversion
  `ex: IEEE.std_logic_unsigned`
- structured types : ***record***
- arrays : general implementation
- pointers : type ***access***
- procedural constructs : style Ada
  `ex: if (...) then ...; ...; ...; end if;`
- case insensitive

More general language (but implementations may be partial)

Verilog

- no redundancy

- No explicit support for multiple architectures in an entity
- predefined logic type and states

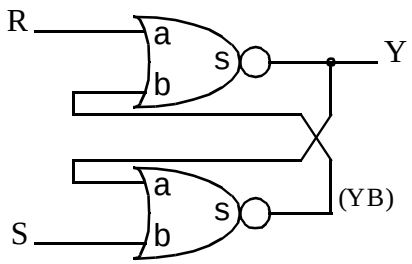
- predefined strengths and resolutions

- built-in implicit integer-vector conversion

- structured types : none
- arrays : only vector arrays (RAM, ROM)
- pointers : no
- procedural constructs : style C - Pascal
  `ex: if (...) begin ...; ...; ...; end`
- case sensitive

More user-friendly language

Structural view : exemple



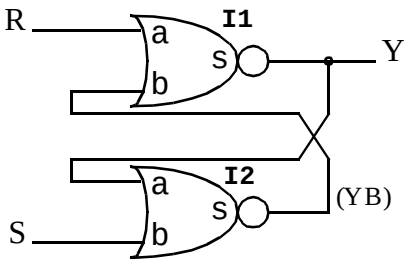
VHDL

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

entity rs_latch is
   port( R, S : in std_logic;
         Y :    out std_logic );
end rs_latch;

architecture circuit of rs_latch is
   component LibNo2
      port( s : out std_logic;
            a, b : in std_logic );
   end component;

   signal Y_port, YB : std_logic;
```

Verilog

```
module rs_latch( R, S, Y );
input R, S;
output Y;

wire YB; // unfortunately optional
```

Explicit connexion




```
begin
Y <= Y_port;
I1 : LibNo2
     port map( a => R,  b => YB, s => Y_port );
I2 : LibNo2
     port map( s => YB, a => Y_port,  b => S );

end circuit;
```

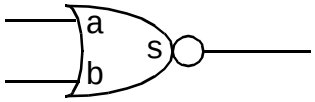
```
    LibNo2 I1 ( .a(R),  .b(YB), .s(Y) );

    LibNo2 I2 ( .s(YB), .a(Y),  .b(S) );

endmodule
```

Alternative solution , implicit connection (by port order)

```
begin
Y <= Y_port;
I1 : LibNo2 port map( Y_port, R, YB );
I2 : LibNo2 port map( YB, Y_port, S );

end circuit;
```

```
    LibNo2 I1 ( Y, R, YB );
    LibNo2 I2 ( YB, Y, S );

endmodule
```

Concurrent description (Data flow)



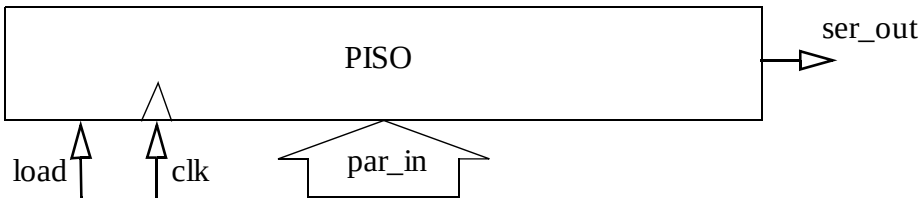
VHDL

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

entity LibNo2 is
   port( s : out std_logic;
         a, b : in std_logic);
end LibNo2;

architecture ARC of LibNo2 is
begin
  s <= not( a or b );
end ARC;
```

Verilog

```
module LibNo2 ( s, a, b );   // NOR gate
  output s;
  input a, b;

  assign s = ~(a|b);
endmodule
```

Procedural description :
“PISO” shift register
(Parallel-IN, Serial-OUT)



VHDL

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

entity piso4 is
port(
    clk, load : in std_logic;
    par_in : in std_logic_vector (3 downto 0);
    ser_out : out std_logic
    );
end piso4;

architecture ARC of piso4 is


signal contenu : std_logic_vector (3 downto 0);
```

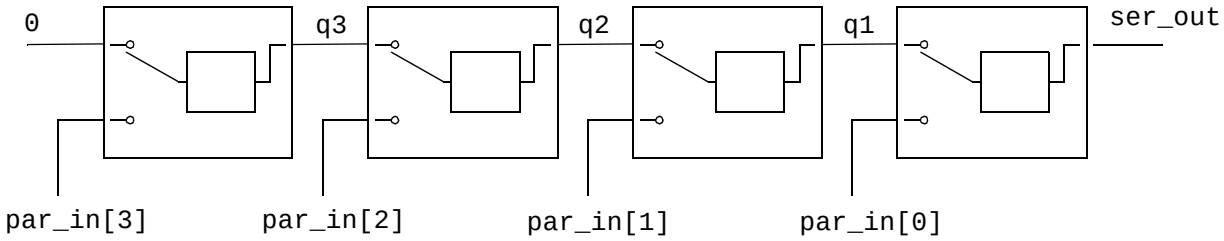
Verilog

```verilog
module piso4( clk, load, par_in, ser_out );

input clk, load;
input [3:0] par_in;
output ser_out;


reg [3:0] contenu;
```

```
begin

process (clk)
begin
if ( clk'event and clk = '1' ) then
   if ( load = '1' )
   then contenu <= par_in;
   else contenu <= ('0' & contenu(3 downto 1));
   end if;
end if;
end process;

ser_out <= contenu(0);

end ARC;
```

```
always @ (posedge clk)
begin

if   ( load )
     contenu <= par_in;                  // load
else
     contenu <= { 1'b0, contenu[3:1] }; // shift

end

assign ser_out = contenu[0];

endmodule
```

This circuit is expected...



Notes on the previous examples :

vector declaration :

```
par_in : in std_logic_vector (3 downto 0);
```

(msb on the left is recommanded)

concurrent assignment

```
s <= not( a or b );
ser_out <= contenu(0);
```

(left member may be **port** or **signal**)
(right member CANNOT reference an output port)

synchronous process (==> infer DFF) :

```
process (clk)
begin
if ( clk'event and clk = '1' ) then
```

procedural assignment (in a process) :

```
contenu <= par_in;
```

(left member may be **port** or **signal**)
(right member CANNOT reference an output port)

Shifting a vector (to the right, unsigned) :

```
('0' & contenu(3 downto 1))
```

(symbol & concatenates vectors)

vector declaration :

```
input [3:0] par_in;
```

(msb on the left is recommanded)

continuous assignement

```
assign s = ~(a|b);
assign ser_out = contenu[0];
```

(left member may be **port** or **net** (**wire**))
(right member can reference any type)

synchronous procedural block (==> infer DFF)

```
always @ (posedge clk)
```

procedural assignement (in a procedural block) :

```
contenu <= par_in;
```

(left member MUST be a **reg**)
(right member can reference any type)

Shifting a vector (to the right, unsigned) :

```
{ 1'b0, contenu[3:1] } or contenu >> 1
```

(symbols {} concatenate vectors)

Inferring a D flip_flop with asynchronous reset

(according to Synopsys documentation)
(triggered by ck rising edge and reset high level)

VHDL

```
signal Q : std_logic;

process( ck, reset )
begin
if ( reset = '1' )
   then Q <= '0';
   else if ( ck'event and ck = '1' )
           then Q <= D;
        end if;
end if;
end process;
```

Verilog

```
reg Q;

always @ ( posedge ck or posedge reset )

   if   ( reset )
        Q = 0;

   else Q = D;
```

Inferring a level sensitive latch

(according to Synopsys documentation)
(loaded by LD high level)

VHDL

```
signal Q : std_logic;

process( D, LD )
begin
  if ( LD = '1' )
     then Q <= D;
  end if;
end process
```

Verilog

```
reg Q;

always @ ( D or LD )

if ( LD )
   Q = D;
```

WARNING : DANGER

Risk of unintentional latch inference in a combinatorial block...

Inferring a tri-state buffer

(according to Synopsys documentation)
(enabled by OE high level)

VHDL

```
signal Q : std_logic;

process( OE, D )
if ( OE = '1' )
   then Q <= D;
   else Q <= 'Z';
end if;
end process;
```

Verilog

```
reg Q;

always @ ( OE or D )
    if   ( OE )
         Q = D;
    else Q = 1'bz;   // high impedance
```

Example : bidirectional register for bus access

Register contains two 8-bit latches, one for writing to the bus, one for reading.
Signal OEN (active low) enables write access to the bus.
Signals LDW et LDR (active high) cause data memorization.

VHDL

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

entity Regbid is
port(
    Din : in std_logic_vector (7 downto 0);
    Dout : out std_logic_vector (7 downto 0);
    Busbi : inout std_logic_vector (7 downto 0);
    OEN, LDW, LDR : in std_logic );
end Regbid;



architecture Lereg of regbid is
signal Lout : std_logic_vector (7 downto 0);
```

Verilog

```
module Regbid
       ( Din, Dout, Busbi, OEN, LDW, LDR );
  input [7:0] Din;
  output [7:0] Dout;
  input  OEN, LDW, LDR;
  inout [7:0] Busbi;

reg [7:0] Busbi; // also inout
reg [7:0] Dout;  // also output


reg [7:0] Lout;
```

```
begin          -- inferring tri-state buffers
process( OEN, Lout )
begin
if ( OEN = '0' )
   then Busbi <= Lout;
   else Busbi <= "ZZZZZZZZ";
end if;
end process;

-- inferring the latches
process ( Din , LDW )
begin
   if (LDW ='1') then Lout <= Din; end if;
end process;

process ( Busbi, LDR )
begin
   if (LDR ='1') then Dout <= Busbi; end if;
end process;

end Lereg;
```

```
                // inferring tri-state buffers
always @ ( OEN or Lout )

    if   ( ~OEN )
         Busbi = Lout;
    else Busbi= 'bzzzzzzzz;  // high impedance



// inferring the latches
always @ ( Din or LDW )

    if ( LDW ) Lout = Din;


always @ ( Busbi or LDR )

    if ( LDR ) Dout = Busbi;
endmodule
```

End of example : bidirectional register for bus access

Notes on the previous examples :

The brackets following the keyword **process** contain the **sensitivity list**. The **process** may be considered as a program which is executed each time one of the signals in the list changes.

```
process( OE, D )
```

A test performed with a **if** allows to restrict the action to the rising edge :

```
if ( ck'event and ck = '1' )
```

Binary constants :

```
'0' -- zero expressed on 1 bit
'1' -- one expressed on 1 bit
'Z' -- high impedance on 1 bit (std_logic)
"ZZZZZZZZ" -- high impedance on 8 bits
```

Assignments :

The right member cannot reference an **out** port, but may reference an **inout** port.

The brackets following the keyword **always @** contain the **sensitivity list** of the procedural block. The block may be considered as a program which is executed each time one of the signals in the list changes. The keyword **or** is used between the elements of the list.

```
always @ ( OE or D )
```

The keyword **posedge** allows to restrict the sensitivity to the rising edge :

```
always @ ( posedge ck )
```

Constantes binaires :

```
1'b0 // zero expressed on 1 bit
1'b1 // one expressed on 1 bit
1'bz // high impedance on 1 bit
8'bzzzzzzzz // high impedance on 8 bits
```

Assignments :

Only **regs** may receive procedural assignments.

outputs and **inouts** ports may be re-declared as **regs**.

```
output [7:0] Dout;
reg [7:0] Dout;  // also output port
```

Synchronous procedural assignments : advanced concepts

Variable assignments or blocking assignments are order-sensitive :

VHDL

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

entity Blk is
port( ck, D : in std_logic;
      X : out std_logic );
end Blk;

architecture ARC of Blk is
begin
  process (ck)
     variable V1, V2 : std_logic;
     begin
     if ( ck'event and ck='1' ) then
        X  <= V2;
        V1 := D;
        V2 := not(V1);
     end if;
  end process;
end ARC;
-- 2 flip-flops (V1 is eliminated) ...
```

Verilog

```
module Blk(ck, D, X);
input ck, D;
output X ;

reg X;



reg V1, V2;
always @ (posedge ck )
begin
  X  = V2;
  V1 = D;
  V2 = ~V1;
end

endmodule
// 2 flip-flops (V1 is eliminated)
```

Synchronous procedural assignments : advanced concepts

Variable assignments or blocking assignments , continued :

VHDL

```
architecture ARC of Blk is
begin
  process (ck)
    variable V1, V2 : std_logic;
    begin
    if ( ck'event and ck='1' ) then
       V1 := D;
       V2 := not(V1);
       X  <= V2;
    end if;
  end process;
end ARC;
-- 1 flip-flop : (V1 and V2 are eliminated)
```

Verilog

```
  reg X;


  reg V1, V2;
  always @ (posedge ck )
  begin
    V1 = D;
    V2 = ~V1;
    X  = V2;
  end


  // 1 flip-flop : (V1 and V2 are eliminated)
```

Synchronous procedural assignments : advanced concepts

Variable assignments or blocking assignments , continued :

VHDL

```
architecture ARC of Blk is
begin
  process (ck)
    variable V1, V2 : std_logic;
    begin
    if ( ck'event and ck='1' ) then
       X  <= V2;
       V2 := not(V1);
       V1 := D;
    end if;
  end process;
end ARC;
-- 3 flip-flops !
```

Verilog

```
  reg X;


  reg V1, V2;
  always @ (posedge ck )
  begin
    X  = V2;
    V2 = ~V1;
    V1 = D;
  end


  // 3 flip-flops !
```

Synchronous procedural assignments : advanced concepts

Signal assignments or non blocking assignments are not sensitive to order (pure RTL)

VHDL

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

entity Blk is
port( ck, D : in std_logic;
      X : out std_logic );
end Blk;

architecture ARC of Blk is
signal V1, V2 : std_logic;
begin
process (ck)
begin
if ( ck'event and ck='1' ) then
   V1 <= D;
   V2 <= not(V1);
   X  <= V2;
end if;
end process;
end ARC;
-- 3 flip-flops whatever the order is
```

Verilog

```
module Blk(ck, D, X);
input ck, D;
output X ;

reg X;

reg V1, V2;


always @ (posedge ck )
begin
  V1 <= D;
  V2 <= ~V1;
  X  <= V2;
end

endmodule
// 3 flip-flops whatever the order is
```

Notes on synchronous procedural assignments : advanced concepts

VHDL

Variables :
Variable assignments allow a coding style which looks like a program.
Variables containing intermediate results may be eliminated by the synthesis process.

Variables are not visible from outside the process.

Signals :
Signal assignments are deferred to the exit of the current timestep, whatever is their position in the code.
Signals behave like registers, coding style is pure RTL (Register Transfer Level).

Verilog

Blocking :
Blocking assignments allow a coding style which looks like a program. Then **regs** behave like variables.
The **regs** containing intermediate results may be eliminated by the synthesis process.

The **regs** are always visible from outside the process.

Non-blocking :
Non-blocking assignments are deferred to the end of the current timestep, whatever is their position in the code.
Then **regs** behave like registers, coding style is pure RTL (Register Transfer Level).

Assignment type is determined by the operator symbol :
- = (equal) for “variable style”
- <= (arrow) for “RTL style”

Note : a given **reg** can receive assignments of only one type (no mixing)

Concurrent models of combinatorial functions :

3 examples of a multiplexer

VHDL

```
-- boolean version
y <= ( i0 and not(s) ) or (i1 and s );

-- conditional expression
y <= i1 when (s = '1') else i0;

-- vectored version (expandable to N inputs)
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity mux2 is
port( i : in std_logic_vector (1 downto 0) ;
      s : in std_logic_vector (0 downto 0);
      y : out std_logic );
end mux2;

architecture lemux of mux2 is
begin
y <= i( CONV_INTEGER(s) );
end lemux;
```

Verilog

```
// boolean version
assign y = ( i0 & ~s ) | (i1 & s );

// conditional expression (C style)
assign y = (s)?(i1):(i0);

// vectored version (expandable to N inputs)

module Mux2( i, s, y );
input [1:0] i;
input s;
output y;

assign y = i[s];
endmodule
```

Procedural models of combinatorial functions : table based

VHDL

```
signal entree : std_logic_vector (2 downto 0);
signal sortie : std_logic_vector (4 downto 0);

process (entree)
begin
case (entree) is
  when  "000" => sortie <= "01001";
  when  "001" => sortie <= "01101";
  when  "010" => sortie <= "11001";
  when others => sortie <= "11111";
end case;
end process;

-- TIP : ALWAYS put an others entry.
```

Verilog

```
wire [2:0] entree;
reg  [4:0] sortie;

always @ (entree)
begin
case (entree)
  'b000 : sortie = 'b01001;
  'b001 : sortie = 'b01101;
  'b010 : sortie = 'b11001;
  default : sortie = 'b11111;
endcase
end

// TIP : ALWAYS put a default entry.
```

Procedural models of combinatorial functions : algorithm

Description is sequential but a combinatorial circuit is expected...

VHDL

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

entity cnt_ones is
port( ent : in std_logic_vector(8 downto 1);
      res : out std_logic_vector(3 downto 1) );
end cnt_ones;

architecture ture of cnt_ones is
begin
  process (ent)
  variable ires : std_logic_vector(3 downto 1);
    begin
    ires := "000";
      for cnt in 1 to 8 loop
          ires := ires + ent(cnt);
      end loop;
    res <= ires;
  end process;
end ture;
```

Verilog

```
module cnt_ones( ent, res );
  input  [8:1] ent;
  output [3:1] res;

reg [3:1] res;
integer cnt;     // internal variable

always @ (ent)

  begin
  res = 0;
  for ( cnt = 1; cnt <= 8; cnt = cnt + 1 )
      res = res + ent[cnt];


  end
endmodule
```

Notes on procedural models of combinatorial functions

Model must match the following conditions :

1. Every signal involved in the computation of the results of the process (or always block) appears in the sensitivity list

2. All the results of the process (or always block) are recomputed in every case of execution of the process (or always block)

Otherwise, unwanted latches appear !

VHDL

Case :
the "**others**" entry helps satisfy condition 2

for loop :
Loop variable is implicit integer.
Assignments must be variable assignments

Vector :
Index must be integer.
Conversion functions are available between integer and vector.
Vector arithmetics require an extension of the std_logic package.

Verilog

Case :
the "**default**" entry helps satisfy condition 2

for loop:
Loop variable must be declared
Assignements must be blocking

Vector :
Index may be a vector.
Conversion between integer and vector is implicit.

Parametrized or generic cell

VHDL

```
library IEEE; use IEEE.std_logic_1164.all;
entity piso is
 generic ( N : integer := 4 );
 port(
     clk, load : in std_logic;
     par_in : in std_logic_vector (N-1 downto 0);
     ser_out : out std_logic );
end piso;
architecture ARC of piso is
signal cont : std_logic_vector (N-1 downto 0);
begin
process (clk)
begin
if ( clk'event and clk='1' ) then
   if ( load = '1' )
      then cont <= par_in;
      else cont <= ( '0' & contenu(N-1 downto 1) );
   end if;
end if;
end process;
ser_out <= contenu(0);
end ARC;
```

Verilog

```
module piso( clk, load, par_in, ser_out );

parameter N = 4;

input clk, load;
input [(N-1):0] par_in;
output ser_out;


reg [(N-1):0] contenu;

always @ (posedge clk)
  begin

  if   ( load )
       contenu <= par_in;
  else
       contenu <= contenu >> 1;
  end

assign ser_out = contenu[0];
endmodule
```

Parametrized or generic cell : instanciation

VHDL

```
architecture ARC of repiso is

component piso
generic ( N : integer := 4 );
port(
    clk, load : in std_logic;
    par_in : in std_logic_vector (N-1 downto 0);
    ser_out : out std_logic );
end component;

begin
piso_inst : piso
  generic map ( N => 32 )
  port map (ck, load, data_in, ser_data );

end ARC;
```

Verilog

```
defparam piso_inst.N = 32;
piso piso_inst ( ck, load, data_in, ser_data );


// alternative form :
piso #(32) piso_inst ( ck, load, data_in,
ser_data );
```

Test Bench (non synthesizable code) : simulating the shift register example «PISO»

VHDL

```
Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_textio.all;
use std.textio.all;

ENTITY test IS -- test entity has no port
END test;

ARCHITECTURE bench OF test IS

COMPONENT piso4
port(
    clk, load : in std_logic;
    par_in : in std_logic_vector (3 downto 0);
    ser_out : out std_logic
    );
end component;

signal clk, load : std_logic;
signal par_in : std_logic_vector (3 downto 0);
signal ser_out : std_logic;
```

Verilog

```
`timescale 1ns / 10ps

module test;  // test module has no port

// input signals to the circuit under test
// will be driven in procedural mode, so
// they must be declared as regs,
// while output signals from the circuit must
// be declared as wires to be driven by it

reg clk, load;
reg [3:0] par_in;
wire ser_out;
```

Test Bench (non synthesizable code) : circuit instance and main sequence

```
constant p : time := 100 ns;    -- clock period

begin
inst : piso4
  PORT MAP( clk, load, par_in, ser_out );


process -- main test sequence
begin
  load <= '0'; par_in <= "0000";
  wait for p;
  load <= '1';
  par_in <= CONV_STD_LOGIC_VECTOR(13,4);
  wait for p;
  load <= '0';
  wait for (p*6);     -- time enough to shift
  load <= '1'; par_in <= "1011";
  wait for p;
  load <= '0';
  wait for (p*6);
  wait;
end process;
```

```
parameter p=100;      // clock period


piso4 inst ( clk, load, par_in, ser_out );


initial  // main test sequence
  begin
  load = 0; par_in = 0;
  #p
  load = 1;
  par_in = 13;   // decimal integer to vector
  #p
  load = 0;
  #(p*6)     // time enough to shift
  load = 1; par_in = 'b1011;
  #p
  load = 0;
  #(p*6)
  $stop;
  end
```

Test Bench (non synthesizable code) : clock generator and text display

```vhdl
process begin   -- clock source
  clk <= '0'; wait for (p/4);
  loop
     wait for (p/2); clk <= not clk;
  end loop;
end process;

-- synchronous text display
process (clk)
variable li: LINE; -- line buffer
begin
if ( clk = '0' ) then
   write( li, string'("load=") );
   write( li, load );
   write( li, string'(" par_in=") );
   write( li, par_in );
   write( li, string'(" ser_out=") );
   write( li, ser_out );
   writeline( OUTPUT, li );
  end if;
end process;

end bench;
```

```verilog
initial          // clock source
begin
  clk = 0; #(p/4)
  forever #(p/2) clk = ~clk;
end


// synchronous text display
always @ (negedge clk)



  $display("load=%b par_in=%b ser_out=%b",
            load, par_in, ser_out );









endmodule
```

Notes on test bench writing

Since the test bench does not have to be synthesizable, all the constructs of the language may be used freely in the test bench.
This makes the test benches much harder to translate from VHDL to Verilog or conversely. This example is aimed at demonstrating a "portable" style.
File I/O is essential in big projects. Here is an example of text output to the simulator console and log file (special file).
Several processes (or procedural blocks) running in parallel may be usefull in test benches, here we have three of them.

VHDL

Time :
The built-in physical type **time** requires specifying the unit with each numerical value (i.e. **ps**, **ns**, **us** etc...)

The statement **wait for** is used for present-time-relative delays.

One-shot process :
The naked **wait** statement at the end of a process prevents its repetition.

Text output :
A two-stage processing is done : text elements are appended to a line buffer (type **LINE** defined in package **std.textio**), then the buffer is written to a file (or to standard output called **OUTPUT**).
Displaying std_logic bit and vector values requires the package **IEEE.std_logic_textio**.

Verilog

Time :
The **`timescale** directive placed before the module header determines the unit used in time values. Its second argument is the claimed time resolution (actual resolution may be better)

The # symbol (hash sign) is used for present-time-relative delays.

One-shot process :
The **initial** keyword creates a procedural block like the **always** keyword, but this block is executed only once.

Text output :
System function **$display** is similar to printf in C language.

Appendix : some synthesizable Verilog 2001 extensions

Procedural model of combinatorial function

In Verilog 95, the sensitivity list should include every input signal so that the synthesis program infers a combinational block (page 24)

In Verilog 2001, the **always @(*)** block is automatically made sensitive to all signals read within the procedure, so that combinational logic is inferred in simulation and synthesis (may be written without parenthesis : **always @***)

Parametrized instance

In Verilog 95, the #() list contains parameter values, to be assigned by order (page 26)

```
piso #(32) piso_inst ( ck, load, data_in, ser_data );
```

In Verilog 2001, an explicit syntax is also allowed, similar to the explicit port connexion (page 5) :

```
piso #(.N(32)) piso_inst ( ck, load, data_in, ser_data );
```