ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Digital Systems Modeling

Chapter 2
VHDL-Based Design

Alain Vachoux
Microelectronic Systems Laboratory
alain.vachoux@epfl.ch

Chapter 2: Table of contents

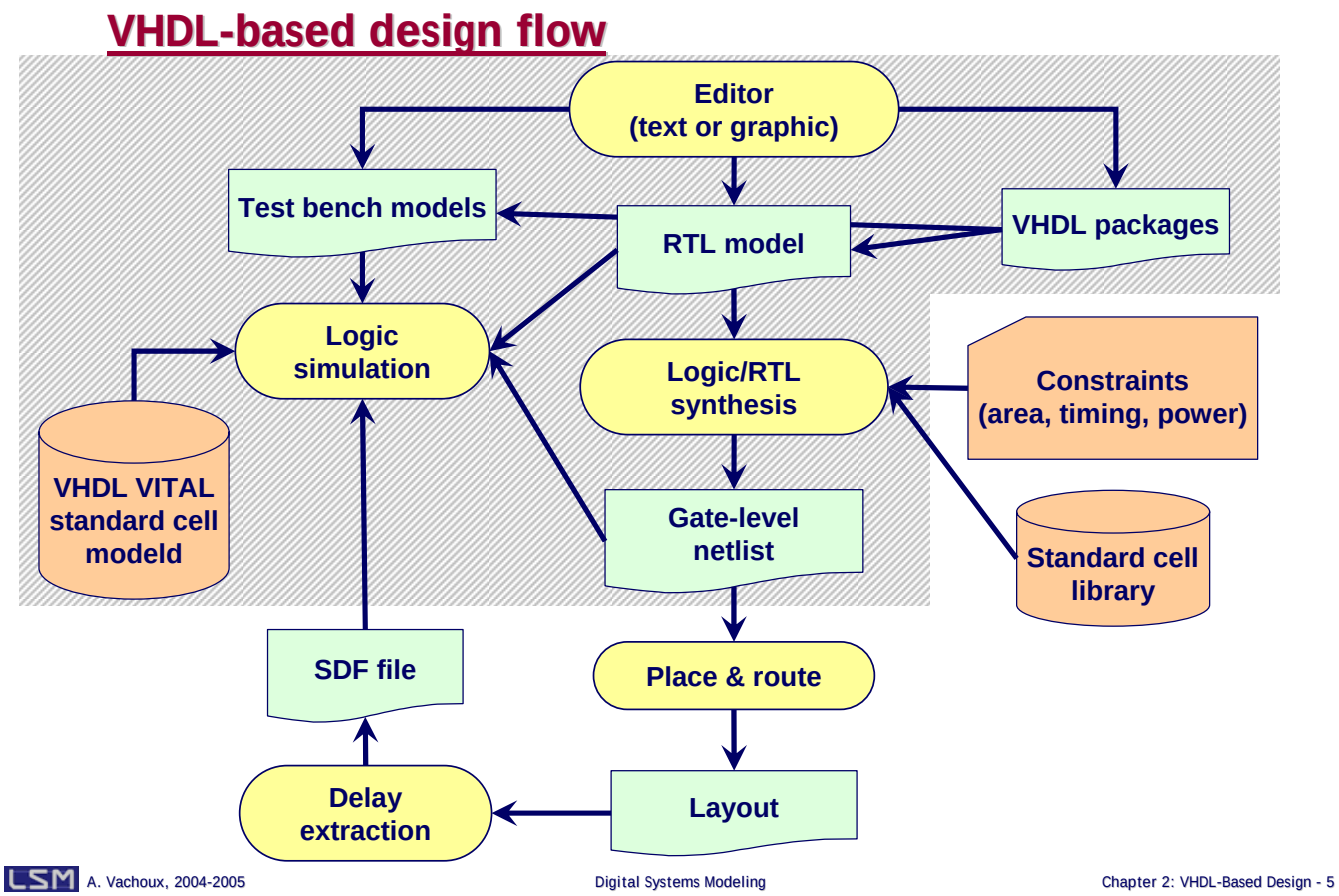
- VHDL overview
- Synthesis with VHDL
- Test bench models & verification techniques

VHDL highlights (1/2)

- **Hardware description language**
  - Digital hardware systems
  - Modeling, simulation, synthesis, documentation
  - IEEE standard 1076 (1987, 1993, 2002)
- **Originally created for simulation**
  - IEEE standards 1164 (STD_LOGIC) and 1076.4 (VITAL)
- **Further adapted to synthesis**
  - Language subset
  - IEEE standards 1076.3 (packages) and 1076.6 (RTL semantics)

VHDL highlights (2/2)

- **Application domain (abstraction levels): Functional -> logic**
- **Modularity**
  - 5 design entities: entity, architecture, package declaration and body, configuration
  - Separation of interface from implementation
  - Separate compilation
- **Strong typing**
  - Every object has a type
  - Type compatibility checked at compile time
- **Extensibility: User-defined types**
- **Model of time**
  - Discrete time, integer multiple of some MRT (Minimum Resolvable Time)
- **Event-driven simulation semantics**



The VHDL-based design flow starts from a description of the system as a RTL model. Complex behavior is described as finite state machines or Boolean equations. The RTL model may use external declarations from standard or user-defined packages. The RTL model can be written using a text editor or using a graphical editor supporting flow charts, finite state machines or dataflow representations.

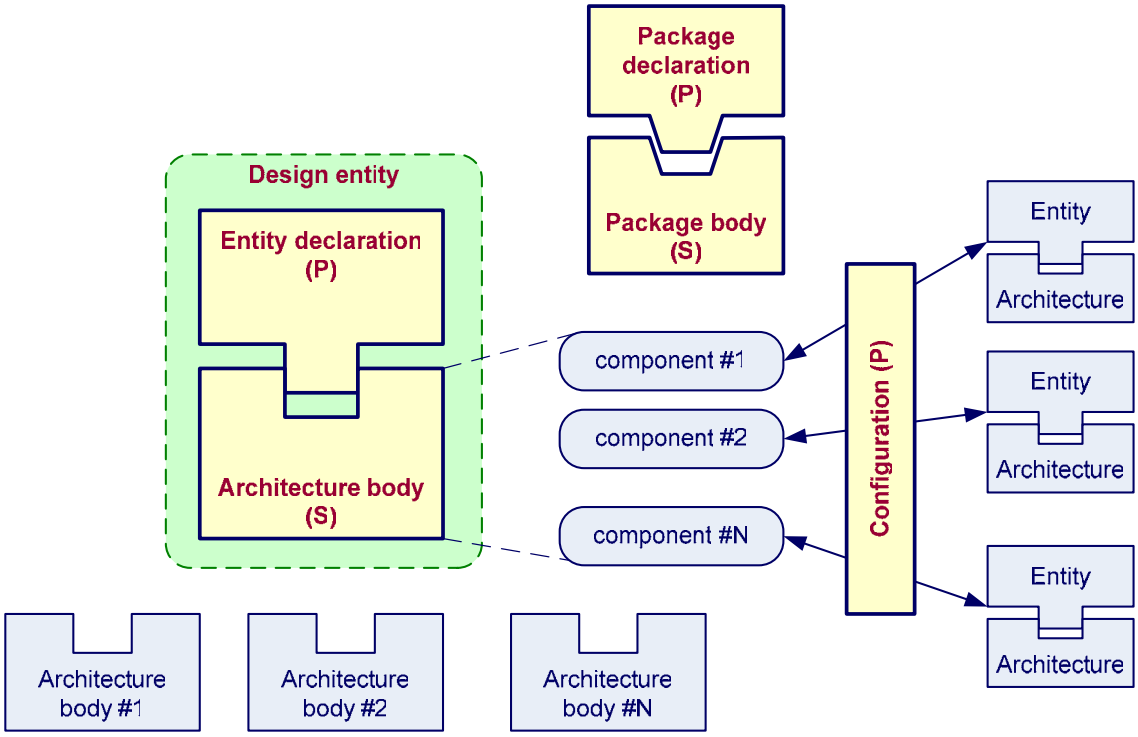
The RTL model can be validated through logic simulation using a VHDL test bench. The test bench declares the design entity to test and stimulus to apply to the unit under test. System functions can then be validated before any realization is actually available.

The RTL model can then be synthesized using a logic synthesizer. The tool is able to derive an optimized gate-level netlist using logic gates from a standard cell library. The optimization is driven by user-defined constraints on area, timings and/or power consumption. The constraints are not included in the VHDL model, but specified separately in the synthesis tool environment. The standard cell library includes information on all the available cells in some technological process (e.g. 0.35µ CMOS): logic functions, areas, timing delays, power consumption. The library format is tool dependent.

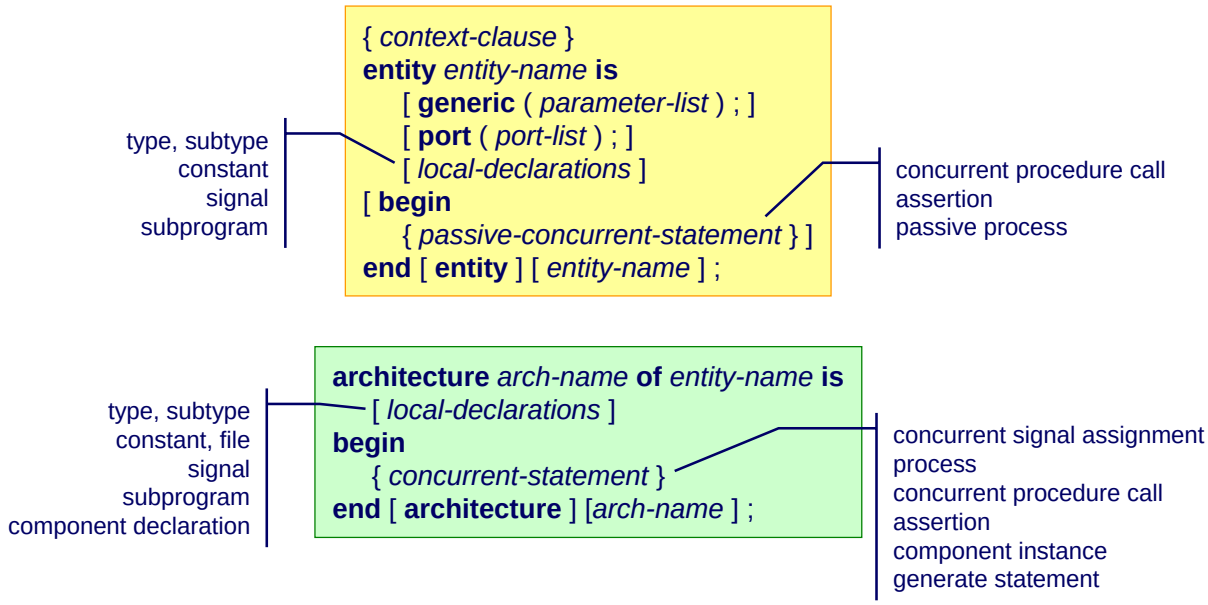
The gate-level netlist can be described in many forms depending on what to do next. A VHDL version of it is usually used for logic simulation. VHDL models of standard cells are provided by the technology provider (foundry or FPGA vendor) in the form of VITAL models. VITAL is an IEEE standard that defines how VHDL models of cells must be written to allow interoperability between different simulation environments. The logic simulation of gate-level netlists now takes care of cell delays and possibly estimated interconnect delays.

The generation of layout is done with a place and route tool that usually requires a description of the gate-level netlist in a different form (e.g. in Verilog, EDIF or XNF). As layout includes true geometrical information, it is possible to extract the values of parasitic R and C elements from wire shapes and to compute timing delays. These delays are stored in the SDF (Standard Delay Format) format and can be back-annotated in VITAL VHDL models of the standard cells. Logic simulation can now take care of more realistic interconnect delays and can be accurate enough to avoid the need to do time consuming circuit-level (SPICE) simulations.

VHDL design units



Design entity



Design libraries

- **Context clause:**

```
library library-name {, …} ;
use selection {, …} ;
```

- **Library names are logical names**
  - Association to physical locations done outside the VHDL model
- **Predefined libraries**
  - WORK
  - STD (incl. STANDARD & TEXTIO packages)
- **Implicit context clause:**

```
library std, work;
use std.standard.all;
```

- **Clause usage**
  - STANDARD package defines the type integer
  - Variable declaration with full path:

```
variable v: std.standard.integer;
```

  - Variable declaration using context clause:

```
variable v: integer;
```

Entity declaration

```
entity entity-name is
    [ generic ( parameter-list ) ; ]
    [ port ( port-list ) ; ]
    [ local-declarations ]
[ begin
    { passive-concurrent-statement } ]
end [ entity ] [ entity-name ] ;
```

```
generic (
    param-name {, …} : param-type [ := default-value ] ;
    …
    param-name {, …} : param-type [ := default-value ]  ) ;
```

```
port (
    [ signal ] signal-name {, …} : mode signal-type ;
    …
    [ signal ] signal-name {, …} : mode signal-type ) ;
```

- **Example: 1-bit full adder**

```
entity add1 is
    generic (
        TP: time := 0 ns);                -- propagation time
    port (
        signal opa, opb, cin: in bit;     -- input operands & carry
        signal sum, cout: out bit);       -- output sum & carry
end entity add1;
```

Architecture body (1/3)

```
architecture arch-name of entity-name is
    [ local-declarations ]
begin
    { concurrent-statement }
end [ architecture ] [arch-name ] ;
```

- **Example: 1-bit full adder, dataflow (concurrent) behavior**
  - Design entity: add1(dfl)

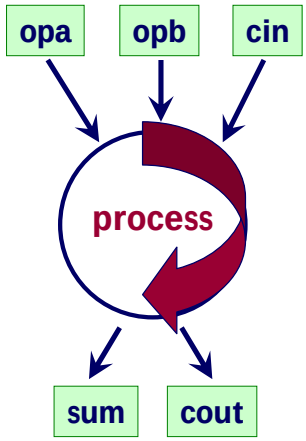
$$S = A \oplus B \oplus Cin$$
$$Cout = (A \bullet B) + (A \bullet Cin) + (B \bullet Cin)$$

```
architecture dfl of add1 is
begin
    sum <= opa xor opb xor cin after TP;
    cout <= (opa and opb) or (opa and cin) or (opb and cin) after TP;
end architecture dfl;
```

Architecture body (2/3)

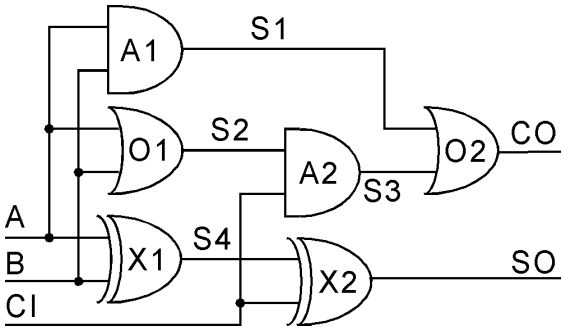
- **Example: 1-bit full adder, sequential behavior**
  - Design entity: add1(algo)

```
architecture algo of add1 is
begin
    process (opa, opb, cin)
        variable tmp: integer;
    begin
        tmp := 0;
        if opa = '1' then tmp := tmp + 1; end if;
        if opb = '1' then tmp := tmp + 1; end if;
        if cin = '1'  then tmp := tmp + 1; end if;
        if tmp > 1  then cout <= '1' after TP;
                    else  cout <= '0' after TP; end if;
        if tmp mod 2 = 0 then sum <= '0' after TP;
                         else sum <= '1' after TP; end if;
    end process;
end architecture algo;
```



Architecture body (3/3)

- **Example: 1-bit full adder, structural model**
  - Design entity: add1(str)
  - Direct instantiation




```
library gates;
architecture str of add1 is
    signal s1, s2, s3, s4: bit;
begin
    A1: entity gates.and2d1(dfl)
            generic map (TPR => TP)
            port map (i1 => opa, i2 => opb, o => s1);
    A2:  entity gates.and2d1(dfl)
            generic map (TPR => TP)
            port map (i1 => s2, i2 => cin, o => s3);
    O1: entity gates.or2d1(dfl)
            generic map (TP)
            port map (opa, opb, s2);
    O2: entity gates.or2d1(dfl)
            generic map (TP)
            port map (s3, s1, cout);
    X1: entity gates.ex2d1(dfl)
            generic map (TPR => TP)
            port map (o => s4, i1 => opa, i2 => opb);
    X2: entity gates.ex2d1(dfl)
            generic map (TP)
            port map (s4, cin, sum);
end architecture str;
```

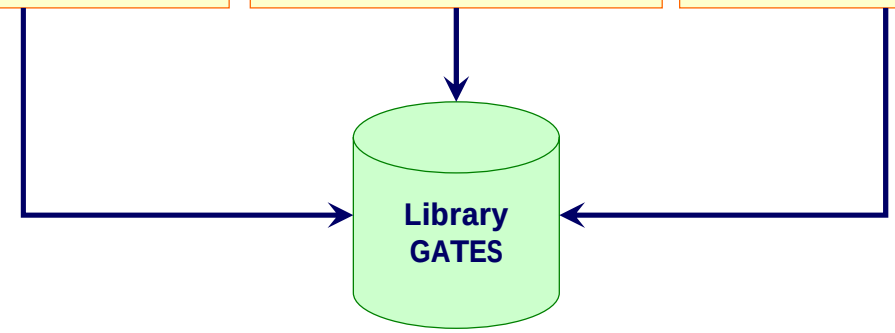
Design library

- **Example: library GATES**

```
entity and2d1 is
    generic (TPR: time := 0 ns);
    port (i1, i2: in bit; o: out bit);
end entity and2d1;
architecture dfl of and2d1 is
begin
    o <= i1 and i2 after TPR;
end architecture dfl;
```

```
entity ex2d1 is
    generic (TPR: time := 0 ns);
    port (i1, i2: in bit; o: out bit);
end entity ex2d1 ;
architecture dfl of ex2d1 is
begin
    o <= i1 xor i2 after TPR;
end architecture dfl;
```

```
entity or2d1 is
    generic (TPR: time := 0 ns);
    port (i1, i2: in bit; o: out bit);
end entity or2d1;
architecture dfl of or2d1 is
begin
    o <= i1 or i2 after TPR;
end architecture dfl;
```



Testbench

- **Example: 1-bit full adder, truth table check**

```
entity tb_add1 is
end entity tb_add1;

architecture bench of tb_add1 is
    signal op1, op2, ci, sum, co: bit;
begin
    UUT: entity work.add1(dfl)
        generic map (TP => 1.2 ns)
        port map (
            opa => op1,
            opb => op2,
            cin  => ci,
            sum => sum,
            cout => co);
```

```
    Stimulus_check: process
        type table_elem is record
            x, y, ci, co, s: bit;
        end record;
        type table is array (0 to 7) of table_elem;
        constant TT: table :=
            (-- x -- y -- ci ------ co -- s --
              ('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'));
    begin
        for i in TT'range loop
            op1 <= TT(i).x;
            op2 <= TT(i).y;
            ci <= TT(i).ci;
            wait for 5 ns;
            assert co = TT(i).co and sum = TT(i).s;
        end loop;
        wait;  -- stop définitif du processus
    end process Stimulus_check;
end architecture bench;
```

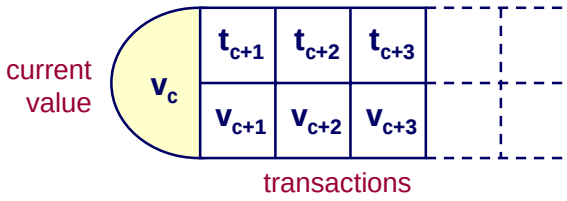
Signals

- **Signal declaration:** **signal** *signal-name* {, ... } : *type* [:= *expression*] ;

- **Examples:**

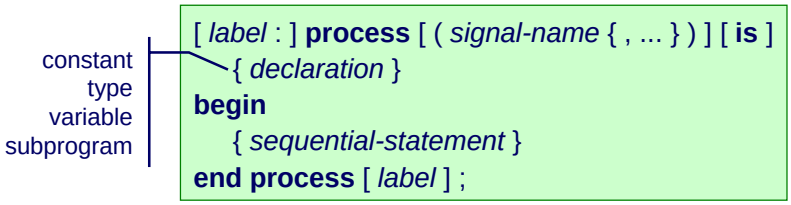
```
signal S: bit_vector(15 downto 0);      -- default initial value = (others => '0')
signal CLK: bit := '1';                 -- explicit initial value
signal reset, strobe, enable: boolean;  -- default initial value = FALSE
```

- **Signal driver:**



Process statement

- **Basic concurrent statement:**



- **Process life cycle:**
  - Created at elaboration time with all its local declarations (e.g. variables)
  - Activated/stopped during simulation (variables retain state)
  - Destroyed at the end of the simulation

- **Not a subprogram!**

Process activation control

- **Either through a sensitivity list:**

```
process (S1, S2, …)
begin
    sequential statements
end process;
```

```
-- equivalent to sensitivity list
process
begin
    sequential statements
    wait on S1, S2;
end process;
```

- **Or through wait statements:**

```
process
begin
    …
    wait …;
    …
end process;
```

```
wait on S1, S2, S3;
```

```
wait for 10 ns;
```

```
wait on S1, S2 until en = '1' for 15 ns;
```

```
wait until clk = '1';
wait on clk until clk = '1';      -- equivalent to previous one
wait on reset until clk = '1';  -- not sensitive to an event on clk
```

```
wait;
wait until next_event;   -- with variable next_event: boolean;
```

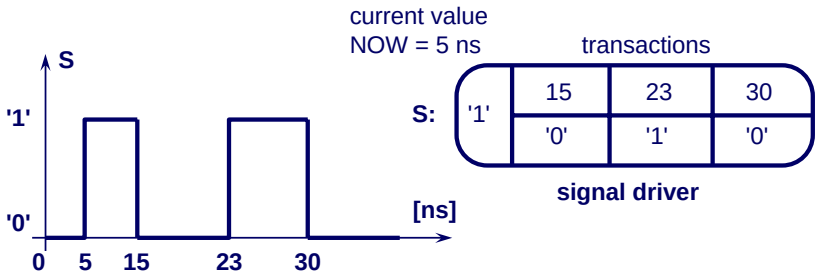
Signal assignment statement

- **Concurrent or sequential statement:**

[*label* :] *signal-name* <= [delay-*mode*] *value-expression* [**after** *time-expression*] {, … } ;

- **Examples:**

```
-- signal S: bit;
stimulus: process
begin
    wait for 5 ns;
    S <= ’1’, ’0’ after 10 ns, ’1’ after 18 ns, ’0’ after 25 ns;
    wait; -- forever
end process stimulus;
```

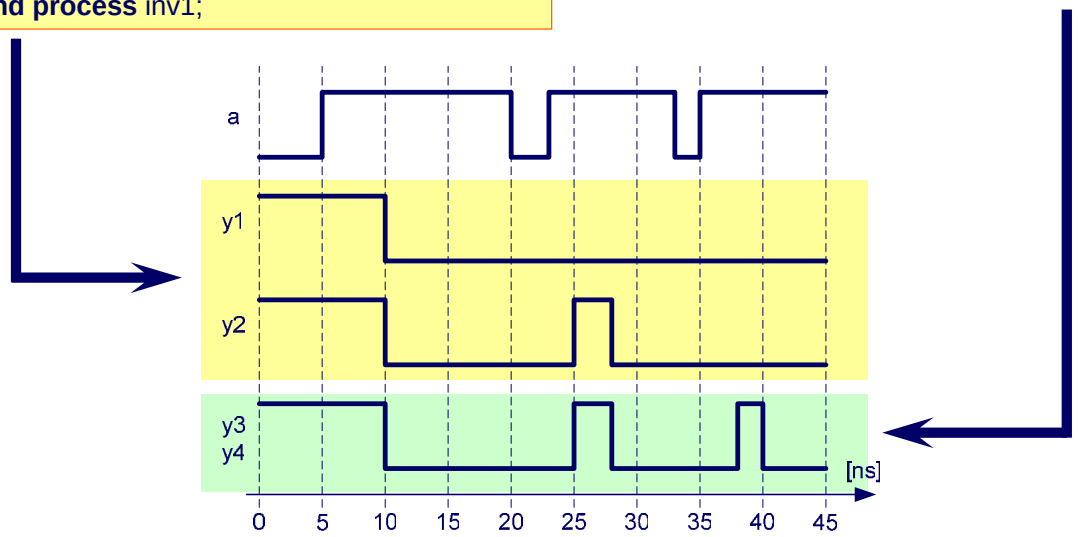



```
-- signal S: bit;
stimulus: process
begin
    wait for 5 ns;
    S <= '1';
    wait for 10 ns;
    S <= '0';
    wait for 8 ns;
    S <= '1';
    wait for 7 ns;
    S <= '0';
    wait;
end process stimulus;
```

Delay modes

```
inv1: process (A) is
begin
    Y1 <= not A after 5 ns;
--  Y1 <= inertial not A after 5 ns;
    Y2 <= reject 2 ns inertial not A after 5 ns;
end process inv1;
```

```
inv2: process (A) is
begin
    Y3 <= transport not A after 5 ns;
    Y4 <= reject 0 ns inertial not A after 5 ns;
end process inv2;
```



Process examples

♦ Clock generator

```
-- signal clk: bit := '0';
clk_gen: process (clk)
begin
    clk <= not clk after 5 ns;  -- 100 MHz
end process clk_gen;
```

♦ Asynchronous element

```
-- signal A, B, Q: bit;
MullerC: process
begin
    wait until A = '1' and B = '1';
    Q <= '1';
    wait until A = '0' and B = '0';
    Q <= '0';
end process MullerC;
```

♦ Latch

```
entity latch is
    port (en, d: in bit; q: out bit);
end entity latch;

architecture bhv of latch is
begin
    process (en, d)
    begin
        if en = '1' then
            q <= d;
        end if;
    end process;
end architecture bhv;
```

♦ Flip-flop

```
entity dff is
    port (clk, d: in bit; q: out bit);
end entity dff;

architecture bhv of dff is
begin
    process
    begin
        wait until clk = '1';
        q <= d;
    end process;
end architecture bhv;
```

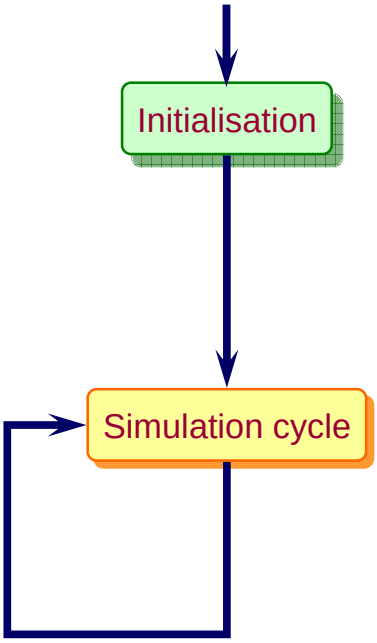
Signal or variable?

```
entity shiftreg is
    generic (W: positive := 8);  -- register width
    port (clk, din: in bit; dout: out bit);
end entity shiftreg;
```

```
architecture sig of shiftreg is
    signal reg: bit_vector(W-1 downto 0);
begin
    process
    begin
        wait until clk = '1';
        reg(W-1) <= din;
        reg(W-2 downto 0) <= reg(W-1 downto 1) ;
        dout <= reg(0);
    end process;
end architecture sig;
```

```
architecture var of shiftreg is
begin
    process
        variable reg: bit_vector(W-1 downto 0);
    begin
        wait until clk = '1';
        dout <= reg(0);
        reg(W-2 downto 0) := reg(W-1 downto 1);
        reg(W-1) := din;
    end process;
end architecture var;
```

Initialization & simulation cycle



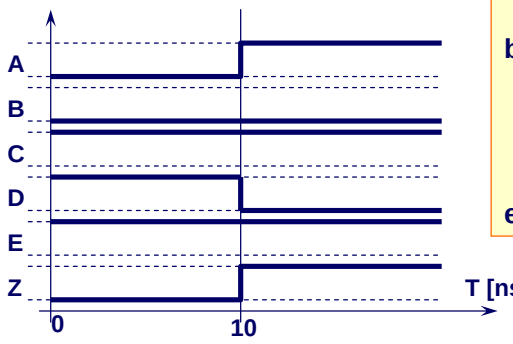
I1. Assign initial values to signals and variables.
I2. Tc = 0 ns, δ = 0.
I3. Execute all processes until they suspend.
I4. Determine next time T_n according to S4.

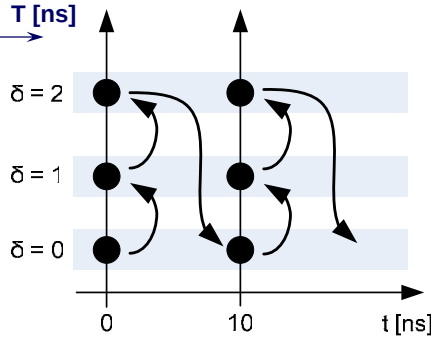
S1. Tc = Tn.
S2. Update signals.
S3. Execute all processes sensitive to updated signals.
S4. Determine next time T_n:
- if pending transactions at current time: $\delta = \delta + 1$ -> S2
- if no more pending transactions or T_n = time'high -> STOP
- else T_n = time of next earliest pending transaction, δ = 0.

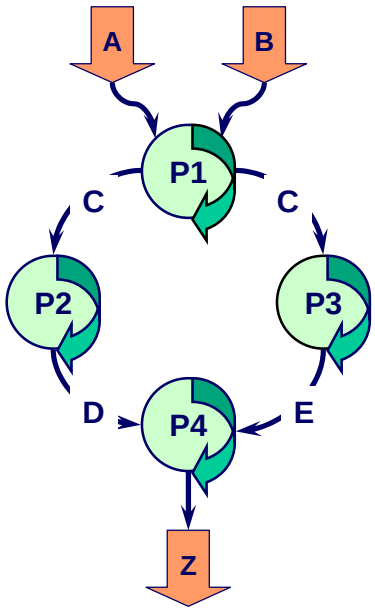
S5. Execute postponed processes.
S6. goto S1.

Zero-delay simulation (delta cycles)

```
entity noteq is
    port (A, B: in bit; Z: out bit);
end entity noteq;

architecture dfl of noteq is
    signal C, D, E: bit;
begin
    P1: C <= A nand B;
    P2: D <= A nand C;
    P3: E <= C nand B;
    P4: Z <= D nand E;
end architecture dfl;
```



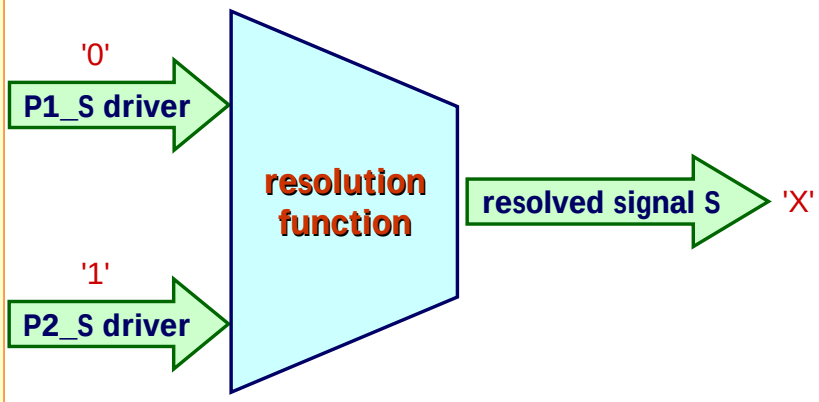




Resolution function

```
-- unresolved types
type ulogic4 is ('X', '0', '1', 'Z');
type ulogic4_vector is array (natural range <>) of ulogic4;
-- resolved types
subtype logic4 is resolve ulogic4;
type logic4_vector is array (natural range <>) of logic4;
```

```
architecture A of E is
    signal S: logic4;
begin
    P1: process begin
        wait for 10 ns;
        S <= '1';
        wait for 20 ns;
        S <= '0';
    end process P1;
    P2: process begin
        wait for 20 ns;
        S <= '0';
        wait for 20 ns;
        S <= '1';
    end process P2;
end architecture A;
```



Default configuration

```
entity inverter is
    generic (TD: delay_length := 0 ns);
    port (iin: in bit; iout: out bit);
end entity inverter;

architecture bhv of inverter is
begin
    iout <= not iin after TD;
end architecture bhv;
```

- **Example: structural buffer**

```
entity buffer is
    port (bin: in bit; bout: out bit);
end entity buffer;

architecture str2 of buffer is
    signal sint: bit;

    component inverter is
        generic (TD: delay_length);
        port (iin: in bit; iout: out bit);
    end component inverter;

begin
    inv1: component inverter
            generic map (TD => 2.5 ns)
            port map (iin => bin, iout => sint);

    inv2: component inverter
            generic map (TD => 3 ns)
            port map (iin => sint, iout => bout);
end architecture str2;
```

Generic parameters

```
entity addn is
    generic (
        TP: time := 0 ns;    -- propagation time
        NB: natural := 8);   -- word size
    port (
        opa, opb: in   bit_vector(NB-1 downto 0);
        cin       : in   bit;
        sum       : out bit_vector(NB-1 downto 0);
        cout      : out bit);
end entity addn;
```

```
architecture dfl of addn is
begin
  process (cin, opa, opb)
    variable ccin, ccout: bit;
    variable result: bit_vector(sum'range);
  begin
    ccout := cin;
    for i in sum'reverse_range loop
      ccin := ccout;
      result(i) := opa(i) xor opb(i) xor ccin;
      ccout := (opa(i) and opb(i))
                 or (ccin and (opa(i) or opb(i)));
    end loop;
    sum <= result after TP;
    cout <= ccout after TP;
  end process;
end architecture dfl;
```

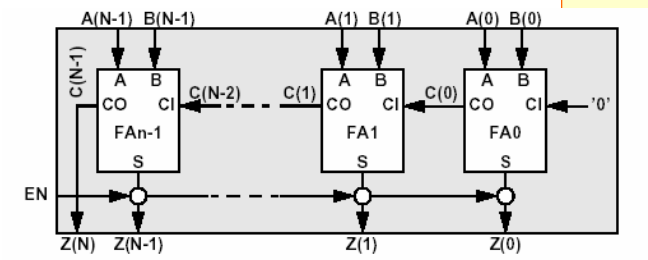
- **Example: generic N-bit adder**

```
entity tb_add32 is
end entity tb_add32;

architecture bench of tb_add32 is
    signal opa, opb, sum: bit_vector(31 downto 0);
    signal cin, cout: bit;
begin
    UUT: entity work.addn(dfl)
                generic map (TP => 2 ns, NB => opa'length)
                port map (opa, opb, cin, sum, cout);
    -- stimulus
    …
end architecture bench;
```

Generate statement

- **Example: structural N-bit adder**

```
entity addn is
    generic (wsize: positive := 8);
    port (
        en  : in   bit;
        a, b: in   bit_vector(wsize-1 downto 0);
        z   : out bit_vector(wsize downto 0));
end entity addn;
```




```
architecture str of addn is
    signal c: bit_vector(wsize-1 downto 0);
begin
    STAGES: for i in wsize-1 downto 0 generate
        signal s_unbuffered: bit;
    begin
        LSB: if i = 0 generate
            FA1: entity work.add1(dfl)
                    port map (opa => a(0), opb => b(0), cin => '0',
                              sum => s_unbuffered, cout => c(0));
        end generate LSB;
        OTHERB: if i /= 0 generate
            FAi: entity work.add1(dfl)
                    port map (opa => a(i), opb => b(i), cin => c(i-1),
                              sum => s_unbuffered, cout => c(i));
        end generate OTHERB;
        OUT_STAGE: process (en)
        begin
            if en = '1' then
                z(i) <= s_unbuffered;
            end if;
        end process OUT_STAGE;
    end generate STAGES;
    z(wsize) <= c(wsize-1);
end architecture str;
```

STD_LOGIC_1164 package (1/2)

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

```
package STD_LOGIC_1164 is
    type std_ulogic is ('U',    -- un-initialized
                        'X',    -- forcing unknown
                        '0',    -- forcing 0
                        '1',    -- forcing 1
                        'Z',    -- high impedance
                        'W',    -- weak unknown
                        'L',    -- weak 0
                        'H',    -- weak 1
                        '-')    -- don't care);
    type std_ulogic_vector is array (natural range <>) of std_ulogic;
    function resolved (s: std_ulogic_vector) return std_ulogic;
    subtype std_logic is resolved std_ulogic;
    type std_logic_vector is array (natural range <>) of std_logic;
    -- overloaded logic operators: and, nand, or, nor, xor, xnor, not
    -- conversion functions: to_bit, to_bitvector, to_stdulogic, to_stdlogicvector, to_stdulogicvector
    -- other fonctions...
end package STD_LOGIC_1164;
```

STD_LOGIC_1164 package (2/2)

```
package body STD_LOGIC_1164 is
    ...
    type stdlogic_table is
        array (std_ulogic, std_ulogic) of std_ulogic;
    constant resolution_table : stdlogic_table := (
        --| 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')    -- | - |);
                                  ...
```

```
    ...
    function resolved (s : std_ulogic_vector) return std_ulogic is
        variable result : std_ulogic := 'Z'; -- default state
    begin
        if s'length = 1 then return s(s'low); -- single driver case
        else
            for i in s'range loop
                result := resolution_table(result, s(i));
            end loop;
        end if;
        return result;
    end function resolved;
    ...
end package body STD_LOGIC_1164;
```

NUMERIC_STD package

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

```
package NUMERIC_STD is
    type unsigned is array (natural range <>) of std_logic;   -- equivalent to unsigned integer
    type signed is array (natural range <>) of std_logic;     -- equivalent to signed integer
    -- abs and "-" unary operators
    -- arithmetic operators: "+", "-", "*", "/", rem, mod
    -- relational operators: "<", ">", "<=", ">=", "=", "/="
    -- shift and rotate operators: sll, srl, rol, ror
    -- logic operators: not, and, or, nand, nor, xor, xnor
    -- conversion functions:
    --      to_integer(arg)
    --      to_unsigned(arg, size)
    --      to_signed(arg, size)
end package NUMERIC_STD;
```

VHDL for synthesis

- **Language subset**
  - All legal VHDL constructs do not have a meaning for synthesis
- **Modeling subset**
  - Synthesis tools recognize specific code templates to infer hardware
- **3 IEEE standards:**
  - IEEE 1164: 9-state logic type `std(u)_logic(_vector)` + logic operators
  - IEEE 1076.3: `unsigned` and `signed` types + logic and arith operators
  - IEEE 1076.6: synthesis semantics

Supported types: enumerated types

- **Types:** bit boolean character std_(u)logic
- **Default encoding:**

```
type state is (idle,  init,  shift,  add,  check);
-- encodage: "000"  "001" "010"  "011" "100"
```

- **Specific encoding, e.g. one-hot:**

```
library synopsys;
use synopsys.attributes.all;

attribute enum_encoding: string;
attribute enum_encoding of state: type is "00001 00010 00100 01000 10000";
--                                         idle   init   shift   add   check
```

- std_(u)logic **(in ieee.std_logic_1164)**
  - Interpreted as 1 bit
  - '**0**', '**L**': low logic level
  - '**1**', '**H**': high logic level
  - '**U**', '**X**', '**W**', '**-**': metalogical states (ignored)
  - '**Z**': high-impedance

```
if enable = '1' then
    request <= ready;
else
    request <= 'Z';
end if;
```


Predefined types bit and boolean are interpreted as single bits. Other enumerated types are encoded. Default encoding is binary encoding with enough bits to represent all enumerated states.

When default encoding is not appropriate (e.g. in finite state machine models), it is possible to use a VHDL attribute declaration do annotate each state with its related encoding word. The VHDL attribute enum_encoding is not predefined and is available in a tool dependent package.
The "one-hot" encoding example shows the 5-bit words defined in a string in the order in which the enumerated states are declared.

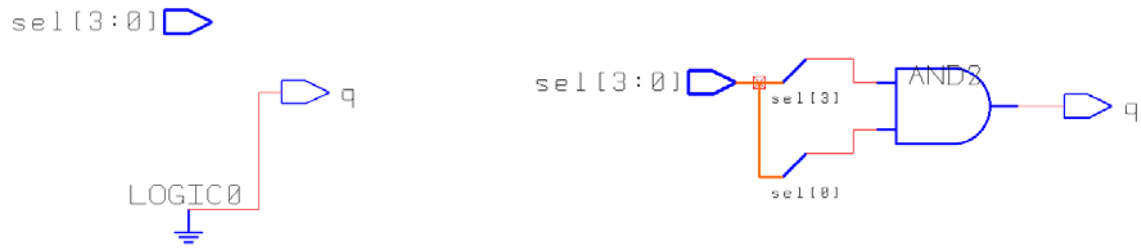
The logic types std_ulogic and std_logic, defined in the STD_LOGIC_1164 standard package, have a specific interpretation for synthesis. Even if they formally have 9 states, they are interpreted in hardware as a single bit. The use of the 'Z' state allows to infer tri-state buffers.

STD_MATCH function

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

entity mux is
  port (
    sel: in  std_logic_vector(3 downto 0);
    q  : out std_logic);
end entity mux;
```

```
architecture dc of mux is
begin
    q <= '1' when sel = "1--1" else '0';
end architecture dc;
```

```
architecture dc_stdm of mux is
begin
    q <= '1' when std_match(sel, std_logic_vector'("1--1")) else '0';
end architecture dc_stdm;
```



The (in)equality check involving metalogical states is always interpreted as being false in synthesis, while it is correctly interpreted in simulation. The standard package NUMERIC_STD defines the function std_match which should be used as a replacement to the "=" operator to ensure correct interpretation in synthesis.

A comparison with the std_match function involving don't care states ('-') infers a circuit in which the don't care bits are discarded.

Integer & array types

- **Integer types:** integer natural positive
  - Infer 32 bit buses by default!
  - Highly recommended to use constrained types

```
-- 7 bits, unsigned
subtype index is natural range 0  to 63;
```

```
-- 8 bits, signed 2's complement
subtype my_byte is integer range -128 to 127;
```

- **Array types:** bit_vector string std_(u)logic_vector unsigned signed
  - One-dimension array with static integer index ranges and scalar or one-dimension array elements
  - "Pack of bits": bit_vector std_(u)logic_vector
  - MSB/LSB: unsigned signed (ieee.numeric_std/_bit)
  - 2-dimension arrays:

```
library ieee; use ieee.numeric_std.all;

subtype word is unsigned(31 downto 0);
    -- MSB = word'left = word(31)
    -- LSB = word'right = word(0)
type register_file is array (0 to 15) of word;
```


Values of type integer of derived subtypes natural and positive are by default interpreted as 32-bit busses. It is therefore highly recommended to constraint the ranges to avoid unnecessary large busses. If values in the range are positive, unsigned values are considered. If they may be negative, signed values in 2's complement are considered.

Only one-dimension array types are supported in synthesis. Index ranges must be static, meaning that the range bounds must be known before simulation starts.

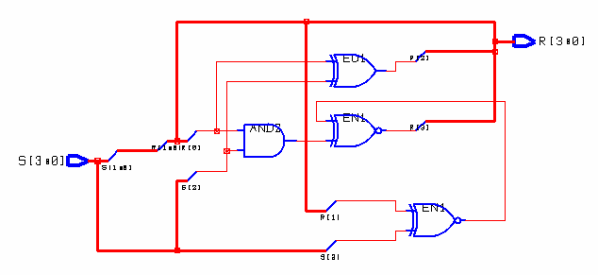
The bit_vector and std_(u)logic_vector are interpreted as mere "packs of bits" without any specific meaning (e.g. no MSB/LSB).

The IEEE 1076.3 standard defines the NUMERIC_BIT and NUMERIC_STD packages that declare the unsigned and signed array types. The difference between the unsigned (signed) types in the packages is the array element type: bit or std_logic.

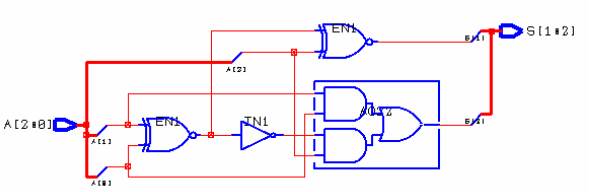
The unsigned type is interpreted (and can be handled) as an unsigned integer. The signed type is intepreted (and can be handled) as a signed integer in 2's complement format. These two types also interpret the bit on the left as the most significant bit (MSB) and the bit on the right as the less significant bit (LSB). The main advantage to use these types is to allow to use arithmetic operations on bit words (which is not possible with the std_(u)logic_vector type).

Constants

```
package cst_pkg is
   subtype int16 is integer range 0 to 15;
end package cst_pkg;

use work.cst_pkg.all;
entity cst is
   port (S: in   int16;
         R: out int16);
end entity cst;

architecture a of cst is
   constant K: int16 := 5;
begin
   R <= S * K;
end architecture a;
```




```
package rom_pkg is
   subtype t_word is bit_vector(1 to 2);
   subtype t_address is natural range 0 to 7;
   type t_rom is array (t_address) of t_word;
end package rom_pkg;

use work.rom_pkg.all;
entity add1b is
   port (A: in t_address; S: out t_word);
end entity add1b;

architecture tt of add1b is
   constant add1b_tt: t_rom := (
            0          => "00",
            1 | 2      => "10",
            3 | 5 | 6  => "01",
            4          => "10",
            7          => "11");
begin
   S <= add1b_tt(A);
end architecture tt;
```



A constant does not infer any hardware so it is highly recommended to use constants as much as possible to minimize area and timings. Constant values are propagated at elaboration time.

Constants can also be used to define ROM content.

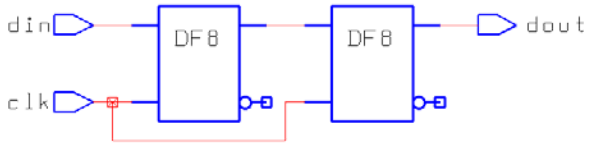
Variables & signals

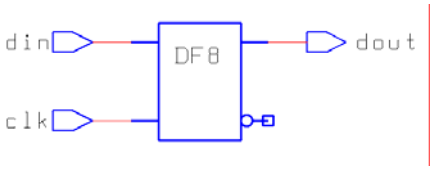
```
entity shiftreg is
   port (clk, din: in bit; dout: out bit);
end entity shiftreg;
```

```
architecture good of shiftreg is
   signal sint: bit;
begin
   process
   begin
      wait until clk = '1';
      sint <= din;
      dout <= sint;
   end process;
end architecture good;
```

```
architecture good2 of shiftreg is
begin
   process
      variable vint: bit;
   begin
      wait until clk = '1';
      dout <= vint;
      vint := din;
   end process;
end architecture good2;
```

```
architecture bad of shiftreg is
begin
   process
      variable vint: bit;
   begin
      wait until clk = '1';
      vint := din;
      dout <= vint;
   end process;
end architecture bad;
```





Variables and signals have different semantics in VHDL and this difference is conserved in synthesis. Both objects can infer a wire, a register or nothing (the object is optimized out).

Recall that a variable has a scope limited to the process or subprogram in which it is defined, while a signal has a global scope in the whole architecture.

The architectures good and good2 correctly describe a two-bit shift register, while architecture bad infers a 1-bit register. Note that the simulation would give the same results.

Initial values

```
entity E is
   port ( ...; clk, rst: in bit; ... );
end entity E;

architecture sync of E is
   signal S: bit_vector(15 downto 0);
begin
   process
   begin
      wait until clk = '1';
      if rst = '1' then
         S <= (others => '0');
         -- + other initializations
      else
         -- normal behavior...
      end if;
   end process;
end architecture sync;
```

synchronous reset

```
entity E is
   port ( ...; clk, rst: in bit; ... );
end entity E;

architecture async of E is
   signal S: bit_vector(15 downto 0);
begin
   process (clk, rst)
   begin
      if rst = '1' then
         S <= (others => '0');
         -- + other initializations
      elsif clk = '1' and clk'event then
         -- normal behavior...
      end if;
   end process;
end architecture async;
```

asynchronous reset

Every VHDL object has an initial value that is either inherited by default from its type or explicitly defined in its declaration. As none of these ways are supported for synthesis, it is required to include explicit initialization code in the model. This is usually done as set/reset behavior and requires the declaration of additional set or reset signals.

A ***synchronous reset*** checks a reset signal at a clock edge. An ***asynchronous reset*** can be done independently of the clock signal.

The same approaches can be used for a set signal.

Synchronous designs usually require a way to put the circuit in a known state at power-up or when it is working.

Operators

- **logical:** or and nor nand xor xnor
- **Relational** [a]**:** = /= < [b] > [b] >= [b] <= [b]
- **Shift & rotate** [c]**:** sll srl sla sra rol ror
- **Addition:** + [b] – [b] & [d]
- **Unary:** + –
- **Multiplication:** * [b],[e],[f] / [g],[h] mod [g] rem [g]
- **Others:** ** [i] abs not

VHDL operators are supported for synthesis with some limitations:

(a) Result is of type boolean.

(b) Can be shared with another operator of the same kind and same priority level.

(c) Introduced in VHDL-1993. The IEEE 1076.6 standard mentions that they are not supported for synthesis and that the functions SHIFT_LEFT, SHIFT_RIGHT, ROTATE_LEFT and ROTATE_RIGHT from packages NUMERIC_BIT/_STD should be used instead. Synopsys tools however do support them.

(d) Concatenation operator '&' can be used to emulate shift and rotate operators.

(e) In general infers a combinational circuit. The inference mechanism depends on the tool (e.g. Synopsys' DesignWare).

(f) If the right operand is a multiple of 2, infers a simple left shifted register.

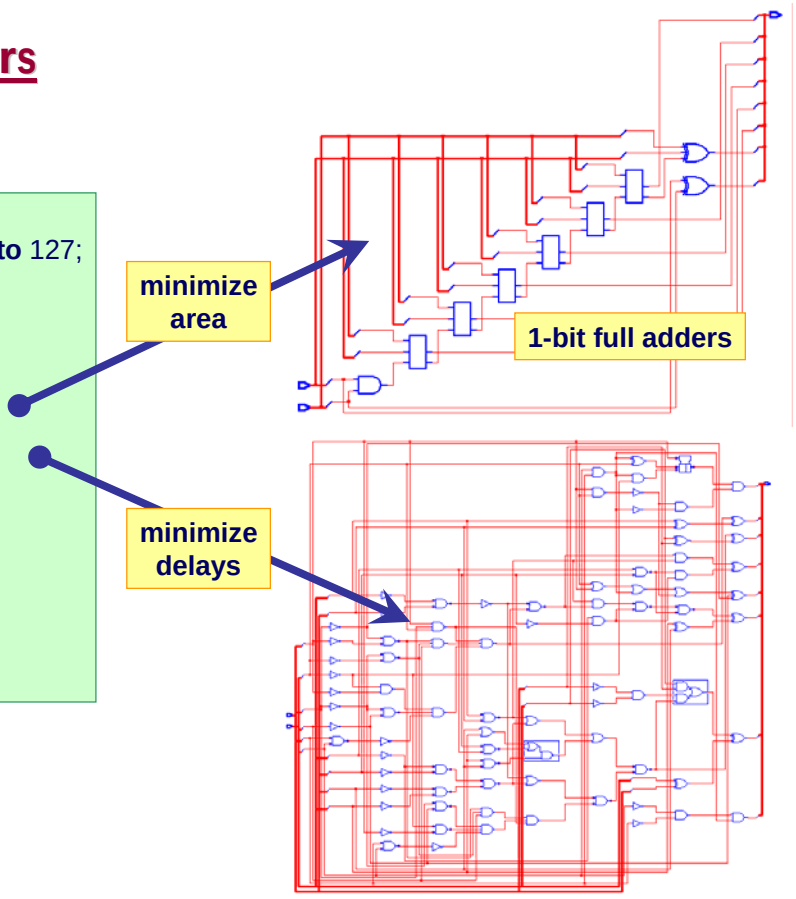
(g) Right operand must be a power of 2.

(h) If the right operand is a multiple of 2, infers a simple right shifted register.

(i) Powered operand must be a constant equal to 2.

Arithmetic operators

```
package add_dw_pkg is
   subtype int8 is integer range -128 to 127;
end package add_dw_pkg;

use work.add_dw_pkg.all;
entity add_dw is
   port (
        A, B: in int8;
        Z: out int8);
end entity add_dw;

architecture dfl of add_dw is
begin
   Z <= A + B;
end architecture dfl;
```

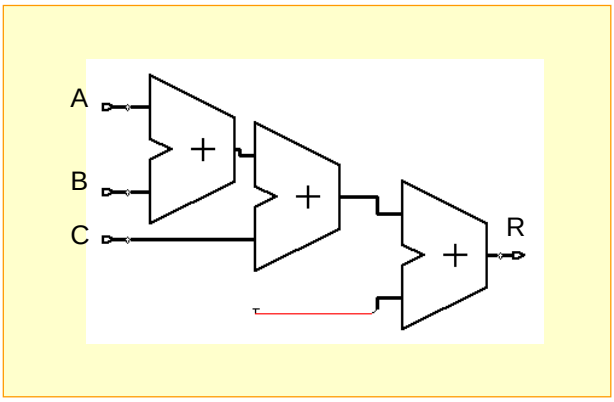


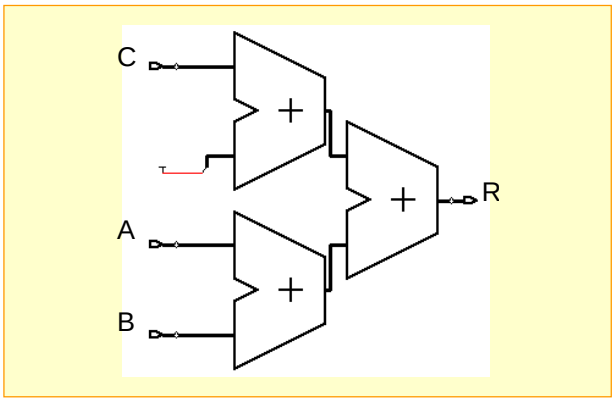
The interpretation of arithmetic operators in synthesis use advanced techniques to select the proper architecture that meets area or timing constraints. Synthesis tools have libraries of synthesizable VHDL models of arithmetic operators (adders, multipliers, etc.) with several architectures.

The simple "+" operator can therefore be mapped to either a serial, small area, carry propagation adder, or to a parallel fast carry look-ahead adder.

The multiply operator "*" usually infers a combinational circuit.

Operator grouping





R <= A + B + C + 1

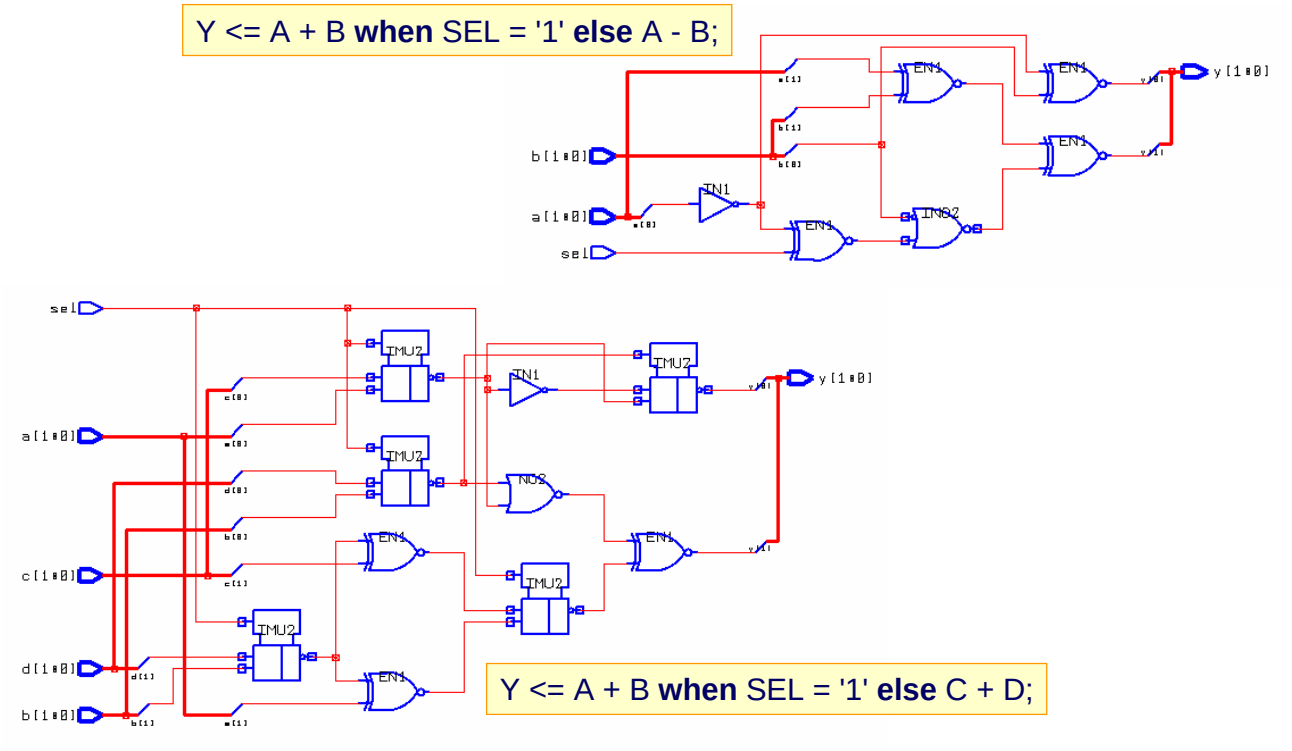
R <= (A + B) + (C + 1)

R <= ((A + B) + C) + 1)

Grouping terms using parentheses allow for overriding the default priority between operator executions. It also allows to infer circuits with different performances.

The given structures are obtained after elaboration but before actual technology mapping took place.

Resource sharing (1/2)



Logic synthesis tools are able to exploit possible sharing of resources such as adders if the VHDL model is written properly. “+”, “-”, “*” and “/” operators can be potentially shared, but only addition and subtraction operators are in practice.

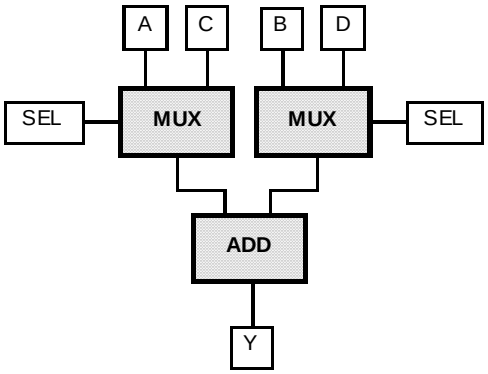
The bottom left figure shows the sharing of one adder for two statements. This is possible because the operands are not the same (actually the sharing would work if only one operand is different). Multiplexers are inferred and make the critical path longer at the benefit of a smaller global area.

The top right figure shows another case of resource sharing where operators are the same, but the operations are not. A single add-subtract component is inferred in this case.

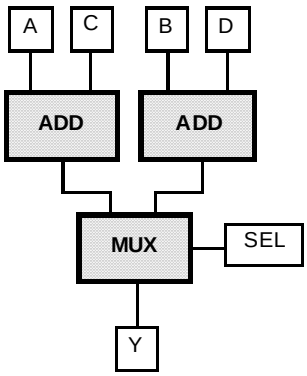
The use of VHDL operators can lead to large (combinational) circuits that could not be optimum in term of area or speed. A typical example is the multiplication operator "*". In these cases it could be worth to use a more detailed model of the operator.

Resource sharing (2/2)

```
-- concurrent statements
MUX1 <= A when SEL = '1' else C;
MUX2 <= B when SEL = '1' else D;
Y <= MUX1 + MUX2;

-- sequential statements
if SEL = '1' then
   MUX1 := A;
   MUX2 := B;
else
   MUX1 := C;
   MUX2 := D;
end if;
Y <= MUX1 + MUX2;
```




```
-- concurrent statements
SUM1 <= A + B;
SUM2 <= C + D;
Y <= SUM1 when SEL = '1' else SUM2;

-- sequential statements
SUM1 := A + B;
SUM2 := C + D;
if SEL = '1' then
   Y <= SUM1;
else
   Y <= SUM2;
end if;
```




```
if SEL = '1' then
   Y <= A + B;
end if;

if SEL /= '1' then
   Y <= E + F;
end if;
```

No possible resource sharing

LSM A. Vachoux, 2004-2005 Digital Systems Modeling Chapter 2: VHDL-Based Design - 42

It is possible to control resource sharing by proper VHDL coding. As an example, the concurrent signal assignment statement:

```
Y <= A + B when SEL = '1' else C + D;
```

can be rewritten to explicitly mention two multiplexers and one adder or two adders and one multiplexerP.

Resource sharing is possible only for one statement at a time and when the statement is included (or is equivalent to) one process.

Process

- **A process infers a combinational circuit if and only if all the following conditions are met:**

  1) The process has a sensitivity list

  2) The process does not declare variables or variables are always assigned before read

  3) All signals that are read in the process are in the sensitivity list

  4) All variables or signals are assigned in every branch of the execution flow (if or case statement)

- **Otherwise a sequential circuit is inferred**
  - Flip-flops are usually required
  - Latches are often not required

The way process statements are written can infer a combinational or a sequential circuit. Combinational circuits have asynchronous behaviors which are only driven by events on signals. Sequential circuits have behaviors which are synchronous to a clock signal and uses flip-flop or latch registers.

The use of concurrent signal assignments always infers combinational circuits.

Clock signal inference

- **Recognized code templates:**
  - In **if** and **wait until** statements

rising_edge (*clock-signal-name*)

falling_edge (*clock-signal-name*)

clock-signal-name 'event **and** *clock-signal-name* = '1'

clock-signal-name 'event **and** *clock-signal-name* = '0'

not *clock-signal-name*'stable **and** *clock-signal-name* = '1'

not *clock-signal-name*'stable **and** *clock-signal-name* = '0'

  - In **wait until** statement

clock-signal-name = '1'

clock-signal-name = '0'

- **Signal name does not convey any meaning for synthesis**
  - Recommended to use meaningful names anyway (e.g. clk)

Wait and if statements

- **Wait statement for inferring behavior sensitive to signal edges**

```
process
   declarations
begin
   wait until clock-edge;      -- must be the first statement in the process
   sequential statements
end process;
```

- **Several wait statements in a process is legal if and only if they relate to the same (clock) signal and the same rising or falling signal edge**

- **If statements for inferring behavior sensitive to signal edges or to signal levels**

```
process (clock-signal-name, …)
   declarations
begin
   do not include any statement here
   if clock-edge then
        sequential statements
   end if;
   do not include any statement here
end process;
```

```
process (clock-signal-name, …)
   declarations
begin
   do not include any statement here
   if signal-level then
        sequential statements
   end if;
   do not include any statement here
end process;
```


wait statements allow for inferring edge-sensitive sequential elements (flip-flops).

Synchronizing a process on different signals or different edges on the same signal is not supported in synthesis. If this is really needed, several processes must be used.

The use of several wait statements in a process sensitive on the same (clock) signal and on the same signal edge is a way to model finite state machines with implicit states, or sequencers.

if statements also allow for inferring level-sensitive sequential elements (latches).

Signal assignment

- **Delay clause is ignored**

S <= '0' **after** 10 ns; → S <= '0';

- **Delay modes are not allowed**
  - **reject**, **inertial**
- **Multiple element waveform is not allowed**

S <= '1', '0' **after** 20 ns, '1' **after** 30 ns; -- error

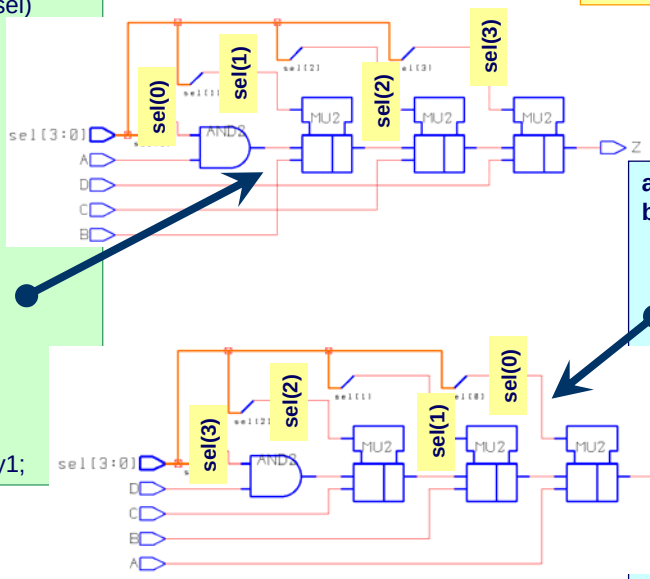
The right-hand part of the signal assignment can be a literal value (e.g. '0' or '1') or any legal expression that evaluates to a value of the same type as those of the assigned signal.

Sequential if statement

- **Implies a pritority**

```
entity ifstmt is
   port (
        A, B, C, D: in bit;
        sel: in bit_vector(3 downto 0);
        Z: out bit);
end entity ifstmt;
```

```
architecture priority1 of ifstmt is
begin
   process (A, B, C, D, sel)
   begin
        Z <= '0';
        if sel(0) = '1' then
             Z <= A;
        end if;
        if sel(1) = '1' then
             Z <= B;
        end if;
        if sel(2) = '1' then
             Z <= C;
        end if;
        if sel(3) = '1' then
             Z <= D;
        end if;
   end process;
end architecture priority1;
```




```
architecture priority2 of ifstmt is
begin
   process (A, B, C, D, sel)
   begin
        Z <= '0';
        if sel(0) = '1' then
             Z <= A;
        elsif sel(1) = '1' then
             Z <= B;
        elsif sel(2) = '1' then
             Z <= C;
        elsif sel(3) = '1' then
             Z <= D;
        end if;
   end process;
end architecture priority2;
```


if statements infer multiplexers or equivalent combinational gates and implies priority. How priority is handled depends on the way the statement is written:

- Several cascaded if statements: last branch in the cascade has the higher priority.
- One if statement with several elsif/else alternatives: the first branch has the highest priority.

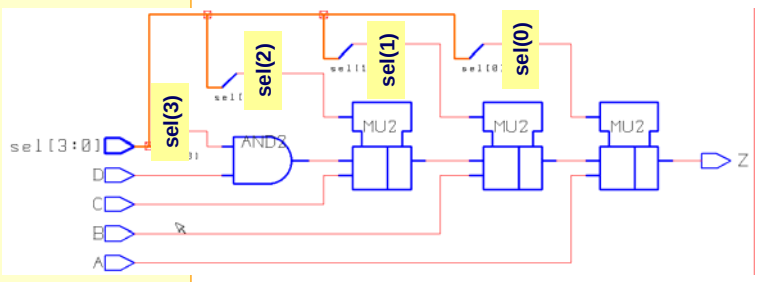
An assignment similar to "Z <= '0'" before the if statement or an else branch prevents inferring unwanted latches:

```
-- priority1
process (...)
begin
    if sel(0) = '1' then
        ...
    end if;
    ...
    if sel(3) = '1' then
        ...
    else
        Z <= '0';
    end if;
end process;
```

```
-- priority2
process (...)
begin
    if sel(0) = '1' then
        ...
    elsif sel(3) = '1' then
    ...
    else
        Z <= '0';
    end if;
end process;
```

Sequential case statement

```
entity casestmt is
   port (
        A, B, C, D: in bit;
        sel: in bit_vector(3 downto 0);
        Z: out bit);
end entity casestmt;

architecture proc of casestmt is
begin
   process (A, B, C, D, sel)
        begin
        case sel is
            when "0001" | "0011" | "0101" | "0111" |
                    "1001" | "1011" | "1101" | "1111" => Z <= A;
            when "0010" | "0110" | "1010" | "1110" => Z <= B;
            when "0100" | "1100" => Z <= C;
            when "1000" => Z <= D;
            when others => Z <= '0';
        end case;
   end process;
end architecture proc;
```



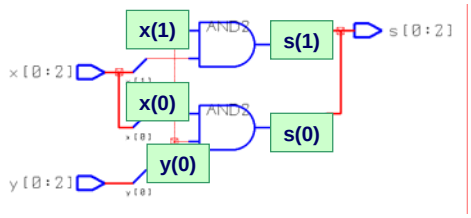
- **Similar to design entity ifstmt(priority2)**

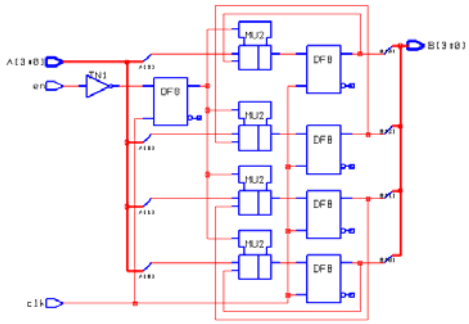
The case statement also infers circuits with multiplexers or equivalent gates. It is interpreted as an if statement with elsif alternatives

The **when others** clause is important to prevent inferring unwanted latches.

Loop statements

```
entity forstmt is
   port (
        x, y: in  bit_vector(0 to 2);
        s    : out bit_vector(0 to 2));
end entity forstmt;

architecture proc of forstmt is
   subtype nat is natural range 0 to 2;
   constant N: nat := 1;
begin
   process (X, Y)
   begin
        for I in X'range loop
             S(I) <= X(I) and Y((I + 1) mod N);
             exit when I = N;
        end loop;
   end process;
end architecture proc;
```





```
library ieee;
use ieee.std_logic_1164.all;
entity whilestmt is
   port (
        A       : in  std_logic_vector(7 downto 0);
        clk, en : in  std_logic;
        B       : out std_logic_vector(7 downto 0));
end entity whilestmt;

architecture rtl of whilestmt is
begin
   process begin
        while (en = '1') loop
             wait until clk'event and clk = '1';
             B <= A;
        end loop;
        wait until clk'event and clk = '1';
   end process;
end architecture rtl;
```


Loop statements (**loop**, **for**, **while**) are supported for synthesis with some restrictions.

The **for** or **while** statement with static iteration limits duplicates the code in the loop (loop unrolling). Premature loop exits are supported.

There are cases for which a loop statement could be avoided. For example, if we have the following signal declarations:

```
signal S1, S2: bit_vector(1 to 10);
```

then the loop statement:

```
for i in S1'range loop
    S2(i) <= S1(i);
end loop;
```

could be rewritten in the more compact form as:

```
S2 <= S1
```

When iteration limits are not static, memory elements have to be inferred to store the loop index. A wait statement is therefore required in the loop body.

The infinite loop statement **loop** … **end loop** is interpreted as a **while** loop whose condition is always true.

Subprograms

- **Do not infer any structural hierarchy (≠ components)**
- **A function call always infers a combinational circuit**
  - Resolution functions and conversion functions are ignored
- **A procedure call infers a combinational circuit if and only if:**

  1) Its arguments are of mode in or out

  2) It does not include any wait statement

  3) It does not have side effects

  Otherwise it infers a sequential circuit

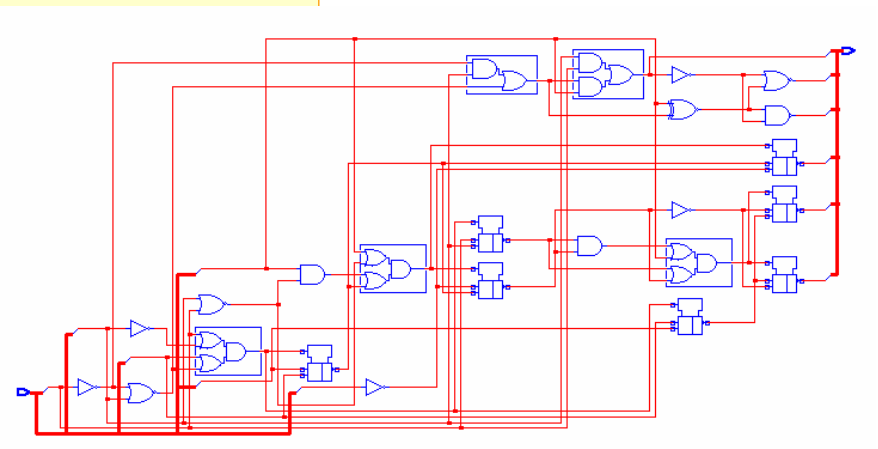
A function call always infers a combinational circuit since it can only appears in an expression.

A procedure call can infer either a combinational or a sequential circuit. Thisis valid for both concurrent and sequential forms of the procedure call.

Procedure

```
architecture a of procstmt is
begin
   process (inar)

      procedure swap (d: inout darray; l, h: in postitive) is
         variable tmp: data;
      begin
         if d(l) > d(h) then
            tmp := d(l);
            d(l) := d(h);
            d(h) := tmp;
         end if;
      end swap;

      variable tmpar: darray;
   begin
      tmpar := inar;
      swap(tmpar,1,2);
      swap(tmpar,2,3);
      swap(tmpar,1,2);
      outar <= tmpar;
   end process;
end architecture a;
```

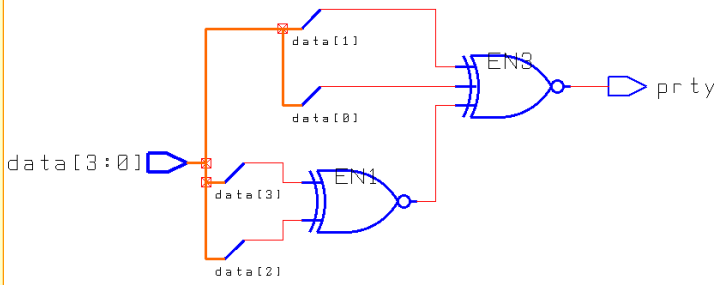
```
package proc_pkg is
   subtype data is integer range 0 to 3;
   type darray is array (1 to 3) of data;
end package proc_pkg;

use work.proc_pkg.all;
entity procstmt is
   port (
      inar  : in   darray;
      outar: out darray);
end entity procstmt;
```



Function

```
entity parity_check is
   generic (NBITS: positive := 8);
   port (
        data: in  bit_vector(NBITS-1 downto 0);
        prty: out bit);
end entity parity_check;
```

```
architecture func of parity_check is
begin
   process (data)

      function parity (bv: bit_vector) return bit is
         variable result: bit;
      begin
         result := '0';
         for i in bv'range loop
              result := result xor bv(i); -- odd parity
         end loop;
         return result;
      end function parity;

   begin
      prty <= parity(data);
   end process;

end architecture func;
```



Concurrent statements

- **Signal assignment always infers combinational circuits**

```
-- conditional form (else is mandatory)
S2 <= (S1 and B) when CMD = '0' else (C or D);
```

```
-- selective form
with CMD select
   S2 <= (S1 and B) when '0' else
         (C or D)     when others;
```

- **Concurrent procedure call always infers combinational circuits**
  - Wait statement not allowed in procedure body
  - Do not infer any structural hierarchy
- **Component instance**
  - Defines a structural hierarchy which is conserved through synthesis
  - Possible operations on components during synthesis:
    - Make instances unique (*uniquify*)
    - Make instances frozen (*don't touch*)
    - Flatten hierarchy (*ungroup*)
- ***generate* statement**
  - Both iterative and conditional forms are supported
  - Local declarations not supported

The structural hierarchy implied by component instances is conserved through synthesis. It is recommended to use components at the RTL level to ease the management of complex designs. This also eases the definition of timing constraints to critical internal parts of the design.

All instances of the same component usually refer to the same component description. It is required to make each instance unique *uniquify*) to allow individual optimization of each instance.

A component may be synthesized separately and then made frozen (*don't touch*) when synthezising one level up in the hierarchy.

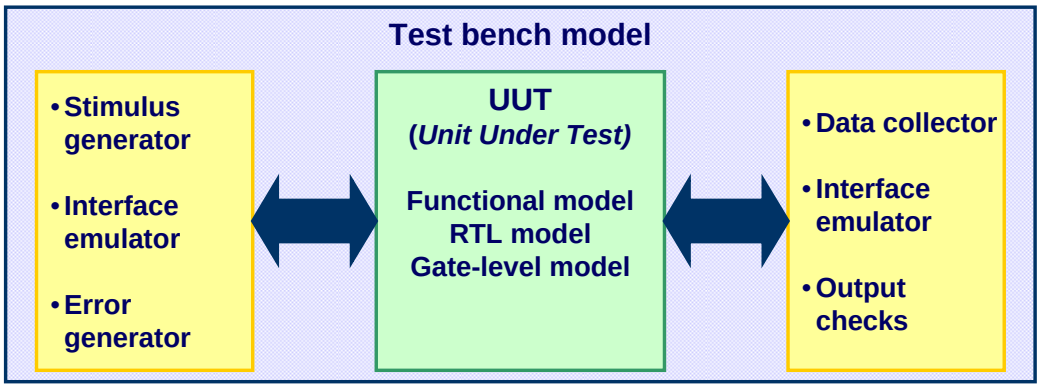
The hierarchy can be flattened (*ungrouped*) during synthesis to allow further optimization across component boundaries.

Miscellaneous

- **Generic parameters**
  - Of type integer or derived
  - Of an enumerated type
- **Configurations**
  - Default configuration only
  - (Direct instantiation)

Default configuration means that there is a component declaration that has exactly the same signature as the entity declaration of a design entity in the working library. Signature includes the signal names, modes and types.

Test bench model




```
entity tb_xxx is
end entity tb_xxx;

architecture bench of tb_xxx is
   …
begin
   UUT: entity work.E(A) port map (…);
   stimulus: process begin
        …
   end process stimulus;
   verification: process begin
        …
   end process verification;
end architecture bench;
```


A test bench model aims at validating a functional, RTL or gate-level model. The kind of valifdation depends on the abstraction level of the unit under test:

- Functional model: interface behavior, communication protocol.
- RTL model: design architecture, control and data parts.
- Gate-level model: timings.

A test bench model can be written in VHDL. Three components may be identified:

- A ***stimulus generator*** whose task is to define the stimulus to apply to the unit under test. Stimulus can be defined in VHDL or in a format closer to the targetted application (e.g. in assembly language or C). In the latter case, the generator has to translate abstract stimulus in VHDL and possibly apply interface constraints (e.g. protocol, delays). The generator may also explicitly introduce errors.
- The ***unit under test*** (UUT).
- A ***collector component*** whose task is to collect output data from the UUT, to possibly translate them into a more readable form and to make checks. Checks can be made either by comparing the output values to ideal values defined in the component, or by comparing output values to other output values generated by a second ideal model stimulated in the same way (e.g. comparing the outputs of a RTL model and a gate-level model).

The unit under test is instantiated as a component. The stimulus and the collector components may be instantiated as components or defined as processes.

More details on test bench modeling and verification methods can be found in [Bergeron00].

Test bench for a 1-bit adder (1/5)

```
entity tb_add1 is
end entity tb_add1;

architecture bench1 of tb_add1 is
    signal op1, op2, ci, sum, co: bit;
begin
    UUT: entity work.add1(dfl)
        generic map (TP => 1.2 ns)
        port map (opa => op1, opb => op2,
                  cin  => ci, sum => sum,
                  cout => co);
    Stimulus_check: process
        procedure check (
            op1, op2, ci, co, sum: in bit;
            exp_co, exp_sum: in bit) is
        begin
            assert co = exp_co and sum = exp_sum
                report "Error for (op1, op2, ci) = (" &
                    bit'image(op1) & "," & bit'image(op2) &
                    "," & bit'image(ci) & ")" & LF &
                    "(co, sum) = (" & bit'image(co) & "," &
                     bit'image(sum) & ") / expected: (" &
                     bit'image(exp_co) & "," &
                     bit'image(exp_sum) &")"
                severity error;
        end procedure check;
        …
        …
    begin
        op1 <= '0'; op2 <= '0'; ci <= '0';
        wait for 5 ns;
        check(op1, op2, ci, co, sum, '0', '0');

        op1 <= '0'; op2 <= '0'; ci <= '1';
        wait for 5 ns;
        check(op1, op2, ci, co, sum, '0', '1');
        …
        op1 <= '1'; op2 <= '1'; ci <= '1';
        wait for 5 ns;
        check(op1, op2, ci, co, sum, '1', '1');

        wait;  -- wait forever
    end process Stimulus_check;

end architecture bench1;
```

```
# ** Error: Error for (op1, op2, ci) = ('0','0','1')
# (co, sum) = ('1','1') / expected: ('0','1')
#    Time: 10 ns  Iteration: 0  Instance: :tb_add1
```


The test bench model uses a single process to define stimulus and to check the outputs. The process generates all possible input values in sequence (truth table).

The check procedure allows for verifying whether the simulated output values are equal to expected values. If not, a message is issued.

The verification uses an assert statement that also specifies a severity level. The simulator can be separately configured to react to particular severity level, e.g. to stop simulation when the severity becomes failure or error.

The 'image attribute allows for converting a value of a predefined type into a string value.

Test bench for a 1-bit adder (2/5)

```
architecture bench2 of tb_add1 is
    signal op1, op2, ci: bit;
    signal sum_dfl, sum_str, co_dfl, co_str: bit;
begin

    UUT: entity work.add1(str)
        generic map (TP => 1.2 ns)
        port map (
            opa => op1, opb => op2, cin  => ci,
            sum => sum_str; cout => co_str);

    UREF: entity work.add1(dfl)
        generic map (TP => 1.2 ns)
        port map (
            opa => op1, opb => op2, cin  => ci,
            sum => sum_dfl; cout => co_dfl);
    …
```

```
    …
    Stimulus_check: process
        procedure check (
            op1, op2, ci, co, sum: in bit;
            exp_co, exp_sum: in bit) is
        begin
            …
        end procedure check;

    begin
        op1 <= '0'; op2 <= '0'; ci <= '0';
        wait for 5 ns;
        check(op1, op2, ci, co_str, sum_str, co_dfl, sum_dfl);

        op1 <= '0'; op2 <= '0'; ci <= '1';
        wait for 5 ns;
        check(op1, op2, ci, co_str, sum_str, co_dfl, sum_dfl);
        …
        op1 <= '1'; op2 <= '1'; ci <= '1';
        wait for 5 ns;
        check(op1, op2, ci, co_str, sum_str, co_dfl, sum_dfl);

        wait;  -- wait forever
    end process Stimulus_check;

end architecture bench2;
```


The test bench model uses a second design entity to serve as a reference model (a dataflow model). The procedure check then uses the outputs of the reference model as reference values to compare with the outputs from the unit under test (a structural gate-level model).

Test bench for a 1-bit adder (3/5)

```
architecture bench3 of tb_add1 is
    signal op1, op2, ci, sum, co: bit;
begin
    UUT: entity work.add1(dfl)
        generic map (TP => 1.2 ns)
        port map (opa => op1, opb => op2,
                  cin  => ci, sum => sum,
                  cout => co);
    Stimulus_check: process
        type table_elem is record
            x, y, ci, co, s: bit;
        end record;
        type table is array (0 to 7) of table_elem;
        constant TT: table :=
            (-- x -- y -- ci ------ co -- s --
              ('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'));
        …
```

```
        …
    begin
        for i in TT'range loop
            op1 <= TT(i).x; op2 <= TT(i).y; ci <= TT(i).ci;
            wait for 5 ns;

            assert co = TT(i).co and sum = TT(i).s
                report "Error for (op1, op2, ci) = (" &
                       bit'image(op1) & "," & bit'image(op2) & "," &
                       bit'image(ci) & ")" & LF &
                       "(co, sum) = (" & bit'image(co) & "," &
                       bit'image(sum) & ") / expected: (" &
                       bit'image(TT(i).co) & "," &
                       bit'image(TT(i).s) &")"
                severity error;

        end loop;
        wait;  -- wait forever
    end process Stimulus_check;

end architecture bench3;
```

```
# ** Error: Error for (op1, op2, ci) = ('0','1','1')
# (co, sum) = ('1','1') / expected: ('1','0')
#    Time: 20 ns  Iteration: 0  Instance: :tb_add1
```


The test bench model declares the truth table of the unit to test that includes all possible inputs and their corresponding expected outputs.

The process body applies each input vector every 5 ns and then checks that the simulated output values are equal to expected values. If not, a message is issued.

Test bench for a 1-bit adder (4/5)

```
use STD.textio.all;
architecture bench4 of tb_add1 is
    file add1_tt: text open read_mode is "add1_tt.dat";
    signal op1, op2, ci, sum, co: bit;
begin
    UUT: entity work.add1(dfl)
        generic map (TP => 1.2 ns)
        port map (opa => op1, opb => op2,
                  cin  => ci, sum => sum,
                  cout => co);
    Stimulus_check: process
        procedure check (
            op1, op2, ci, co, sum: in bit;
            exp_co, exp_sum: in bit) is
        begin
            assert co = exp_co and sum = exp_sum
                report "Error for (op1, op2, ci) = (" &
                    bit'image(op1) & "," & bit'image(op2) &
                    "," & bit'image(ci) & ")" & LF &
                    "(co, sum) = (" & bit'image(co) & "," &
                    bit'image(sum) & ") / expected: (" &
                    bit'image(exp_co) & "," &
                    bit'image(exp_sum) &")"
                severity error;
        end procedure check;
        …
```

```
# op1 op2 ci co sum
00000
00101
01001
01110
10001
10110
11010
11111
```

```
        …
        variable tt: bit_vector(1 to 5);
        variable exp_co; exp_sum: bit;
        variable ll: line;
    begin
        readline(add1_tt, ll);  -- en-tête
        while not endfile(add1_tt) loop
            readline(add1_tt, ll);
            read(ll, tt);
            op1 <= tt(1); op2 <= tt(2); ci <= tt(3);
            exp_co := tt(4); exp_sum := tt(5);
            wait for 5 ns;
            check(op1, op2, ci, co, sum, exp_co, exp_sum);
        end loop;
        wait;  -- wait forever
    end process Stimulus_check;

end architecture bench4;
```


The truth table is now read from a file.

Test bench for a 1-bit adder (5/5)

```
use STD.textio.all;
architecture bench5 of tb_add1 is
    file add1_tt: text open read_mode is "add1_tt.dat";
    file flog: text open write_mode is "tb_add1.log";
    signal op1, op2, ci, sum, co: bit;
begin
    UUT: entity work.add1(dfl)
        generic map (TP => 1.2 ns)
        port map (opa => op1, opb => op2,
                  cin  => ci, sum => sum,
                  cout => co);
    Stimulus_check: process
        procedure check (op1, op2, ci, co, sum: in bit;
                         exp_co, exp_sum: in bit) is
        begin
            ...
        end procedure check;
        variable tt: bit_vector(1 to 5);
        variable exp_co; exp_sum: bit;
        variable llr, llw: line;
    begin
        readline(add1_tt, llr);  -- en-tête
        write(llw,
            string'("time -- op1 op2 ci exp_co exp_sum co sum"));
        writeline(flog, llw);
        …
        …
        while not endfile(add1_tt) loop
            readline(add1_tt, llr);
            read(llr, tt);
            write(llw, time'image(now) & string'(" -- ") &
                       bit'image(tt(1)) & " " & bit'image(tt(2)) & " "
                       bit'image(tt(3)) & " " & bit'image(tt(4)) & " " &
                       bit'image(tt(5)));
            op1 <= tt(1); op2 <= tt(2); ci <= tt(3);
            exp_co := tt(4); exp_sum := tt(5);
            wait for 5 ns;
            write(llw, " " & bit'image(co) & " " & bit'image(sum));
            writeline(flog, llw);
            check(op1, op2, ci, co, sum, exp_co, exp_sum);
        end loop;
        wait;  -- wait forever
    end process Stimulus_check;

end architect
```

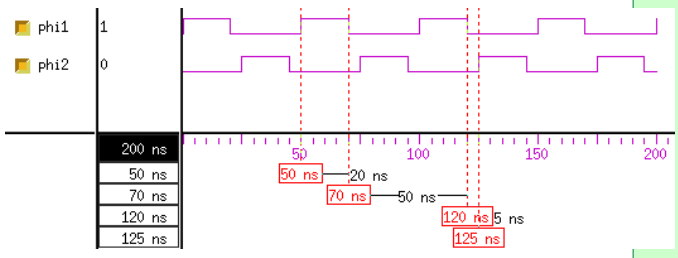
```
time -- op1 op2 ci exp_co exp_sum co sum
0 ns -- '0' '0' '0' '0' '0' '0' '0'
5 ns -- '0' '0' '1' '0' '1' '0' '1'
10 ns -- '0' '1' '0' '0' '1' '0' '1'
15 ns -- '0' '1' '1' '1' '0' '1' '0'
20 ns -- '1' '0' '0' '0' '1' '0' '1'
25 ns -- '1' '0' '1' '1' '0' '1' '0'
30 ns -- '1' '1' '0' '1' '0' '1' '0'
35 ns -- '1' '1' '1' '1' '1' '1' '1'
```


The output of the verification process is now written in a log file.

Clock generation

```
library ieee;
    use ieee.std_logic_1164.all;
architecture bench of tb_xxx is
    constant CLK_PER: time := 20 ns;
    signal clk: std_logic := '0';
begin
    UUT: …
    clk <= not clk after CLK_PER/2;
    Stimulus_check: process
        …
    end process Stimulus_check
end architecture bench;
```

```
library ieee;
    use ieee.std_logic_1164.all;
architecture bench of tb_xxx is
    signal phi1, phi2: std_logic := '0';
    procedure clkgen (
        signal clk: out bit;
        constant Tperiod, Tpulse, Tphase: in time) is
    begin
        wait for Tphase;
        loop
            clk <= '1', '0' after Tpulse;
            wait for Tperiod;
        end loop;
    end procedure clkgen;
    …
begin
    UUT: …
    gen_phi1: clkgen(phi1, Tperiod => 50 ns,
                           Tpulse => 20 ns,
                           Tphase => 0 ns);
    gen_phi2: clkgen(phi2, Tperiod => 50 ns,
                           Tpulse => 20 ns,
                           Tphase => 25 ns);
    …
end architecture bench;
```

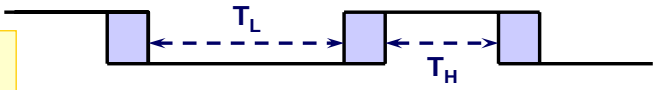


The clock behavior can be defined as a separate process (left) or as a concurrent procedure (right).

The procedure clkgen allows for defining symetrical or asymetrical clocks and then can be used to define nonoverlapping clocks. Furthermore it can be put in a package and reused in several models.

Waveform generation (1/2)

```
library ieee;
    use ieee.math_real.all;

architecture bench of tb_xxx is

    constant PC_MIN: real : = 0.3;    -- % min. value ('0')
    constant PC_MAX: real := 0.3;     -- % max. value ('1')

    constant TL_MIN : time := 5 ns;
    constant TL_MAX: time := 7 ns;

    constant TH_MIN : time := 3 ns;
    constant TH_MAX: time := 5 ns;

    signal S: bit := '0';
begin
    process
        variable seed1: positive := 3812;
        variable seed2: positive := 915;
        …
```



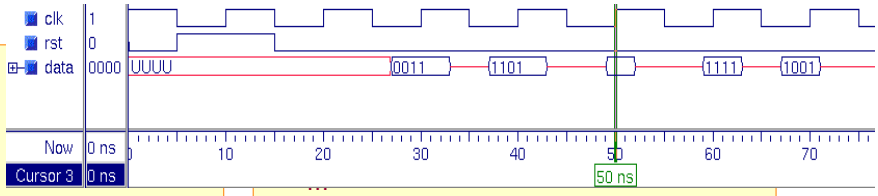

```
        …
        impure function random return real is
            variable rnd: real;
        begin
            uniform(seed1, seed2, rnd);
            if rnd < PC_MIN then
                return 0.0;
            elsif rnd < PC_MIN + PC_MAX then
                return 1.0;
            else
                return rnd;
            end if;
        end function random;
    begin
        S <= '0';
        wait for TL_MIN + (TL_MAX - TL_MIN)*random;
        S <= '1';
        wait for TH_MIN + (TH_MAX - TH_MIN)*random;
    end process;
    …
end architecture bench;
```


The use of the procedure uniform defined in the IEEE standard package MATH_REAL allows for generating waveforms whose states may have random durations between some given minimum and maximum values.

The procedure uniform uses and modifies two arguments called seed1 and seed2 that control the generation of a pseudo-random sequence of values in the open interval]0.0, 1.0[.

The function random modifies the generated random value to meet specific given percentages, namely 30% at minimum duration, 30% at maximum duration, and 40% at a random value between theses limits. The function is declared as ***impure*** as its execution modifies global variables (side effect). The keyword **impure** has been introduced in VHDL-1993.

Waveform generation (2/2)




```
library ieee;
    use ieee.std_logic_1164.all;
architecture bench of tb_proc is
    constant CLK_PER: time := 10 ns;
    constant TSETUP : time := 3 ns;
    constant THOLD  : time := 3 ns;
    signal clk: std_logic := '0';
    signal rst: std_logic;
    signal data: std_logic_vector(3 downto 0);

    procedure do_sync_reset (
        signal clk: in   std_logic;
        signal rst: out std_logic) is
    begin
        rst <= '0'; wait until clk = '0';
        rst <= '1'; wait until clk = '0';
        rst <= '0';
    end procedure do_sync_reset;

    procedure apply_vector (
        constant vector: in   std_logic_vector(3 downto 0);
        signal   data   : out std_logic_vector(3 downto 0);
        constant PER  : in  time := CLK_PER;
        constant TS   : in  time := TSETUP;
        constant TH   : in  time := THOLD) is
    ...
    begin
        wait until clk = '0';
        wait for CLK_PER/2 - TS;
        data <= vector;
        wait until clk = '1';
        wait for TH;
        data <= (others => 'X');
    end procedure apply_vector;

begin
    CLKGEN: clk <= not clk after CLK_PER/2;
    DATAGEN: process begin
        do_sync_reset(clk,rst);
        apply_vector("0011", data);
        apply_vector("1101", data);
        apply_vector("0000", data, 1 ns, 2 ns);
        apply_vector("1111", data, 1 ns);
        apply_vector("1001", data, TH => 1 ns)
        wait;
    end process DATAGEN;
end architecture bench;
```


The use of procedures allows for factorizing the stimulus generation process and possibly to store these procedures in some package to favor reuse.

The procedure do_sync_reset performs a synchronous reset.

The procedure apply_vector assigns a test vector to a data bus with meeting some given setup and hold times.