LIMBAJUL VHDL

OBIECTIVE

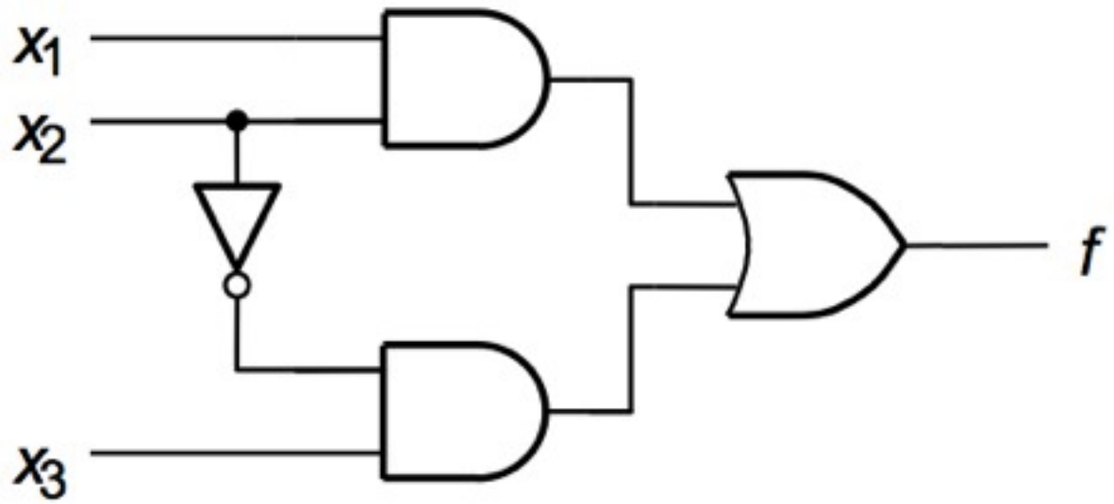
- Concepte de bază ale limbajului
- Metodologii de proiectare
- Scrierea unui cod VHDL foarte simplu

Motivele modelării

- specificarea cerințelor
- documentare
- testare utilizând simulatoarele
- verificare formală
- sinteză

Scopul - evitarea greșelilor

Exemplul 1



X1	X2	X3	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

ENTITATEA - pasul 1

- Declararea semnalellor de I/O utilizând construcția “entity”

```
ENTITY modulul1 IS
        PORT (x1, x2, x3 : IN BIT;
                f: OUT BIT);
END module1;
```

- Semnalele de I/O ale unei entități sunt denumite porturi și se definesc prin cuvântul cheie PORT
- Porturile pot fi de tipul IN sau de OUT
- Semnalul reprezentat de un port are un tip asociat: BIT

ARHITECTURA - pasul 2

- Această construcție este folosită pentru specificarea funcționalității circuitului

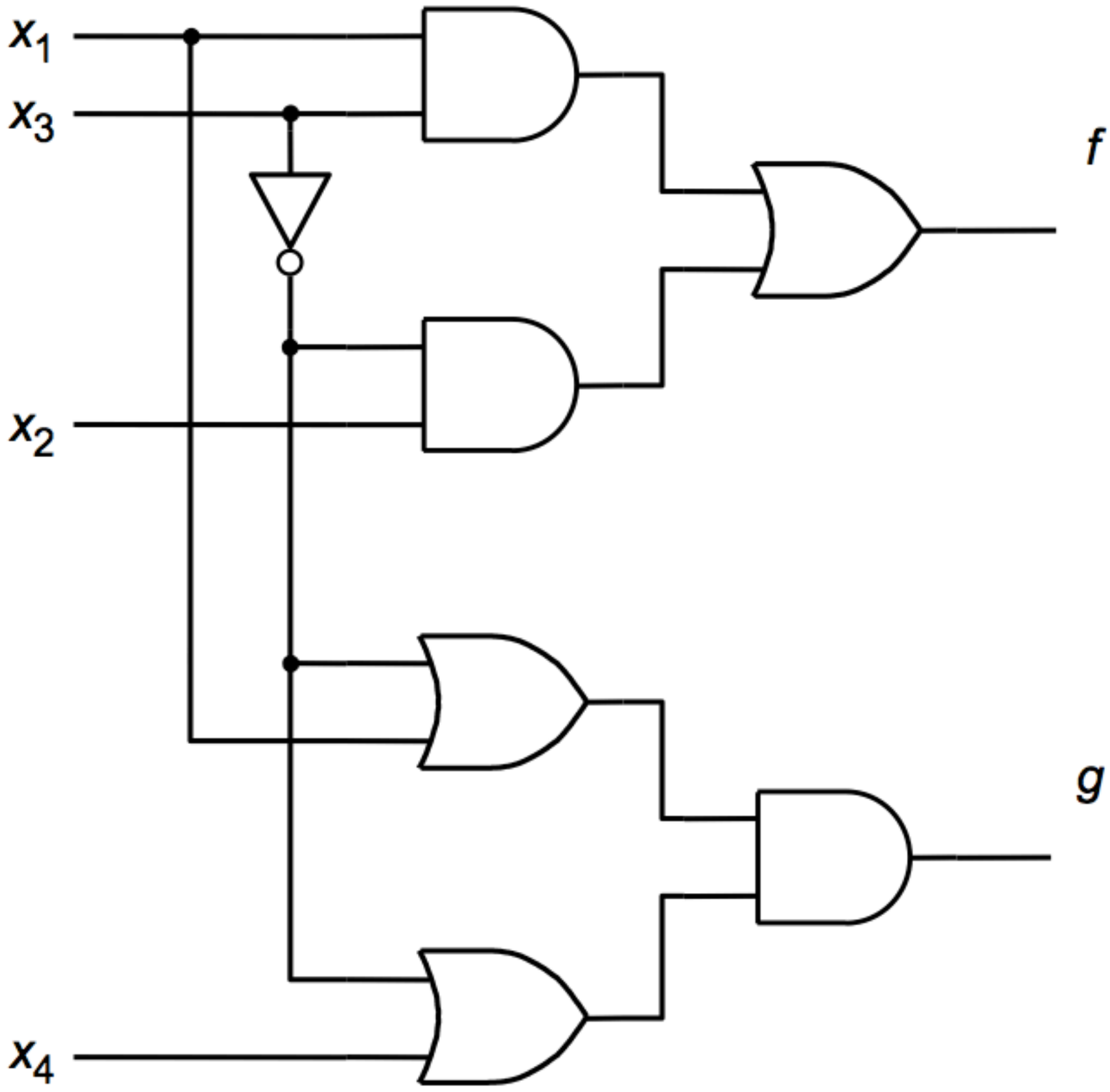
```
ARCHITECTURE functlogica OF module1 IS

BEGIN

    f <= (x1 AND x2)  OR (NOT x2 AND x3)

END functlogica;
```

- VHDL are suport pentru operatorii de tip boolean

- Un modul hardware este definit prin declararea entității care specifică interfața.

- O entitate poate avea una sau mai multe corpuri arhitecturale corespunzătoare. Specificarea arhitecturii utilizate se face într-o unitate denumită *Configuration*

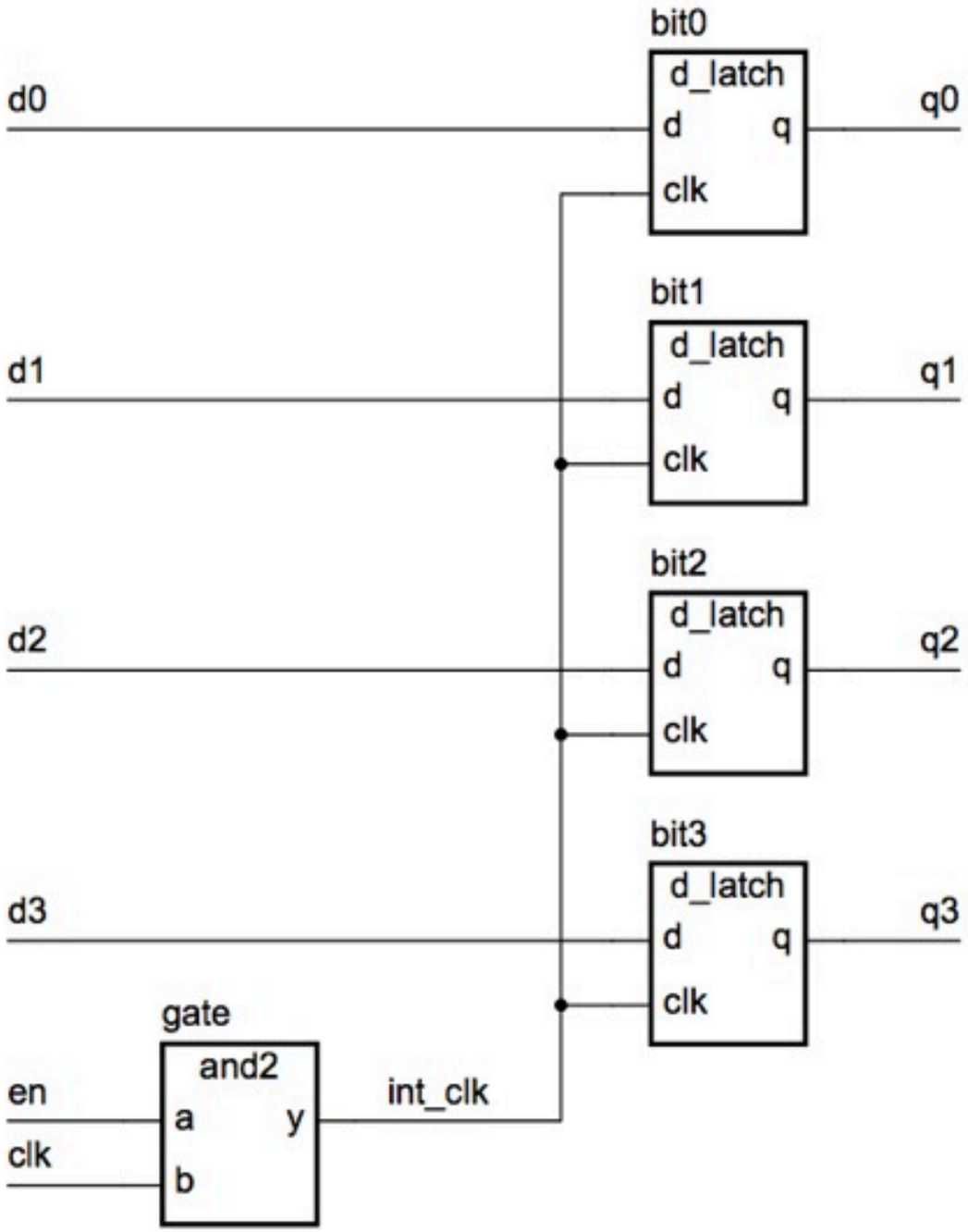
Exemplul 2



SUMATOR COMPLET

```
LIBRARY ieee;

USE ieee.std_logic_1164.all;

ENTITY fulladdser IS

        PORT (cin, x,y : IN STD_LOGIC;

                        s, cout: OUT STD_LOGIC);

END fulladder;

ARCHITECTURE functlogica OF fulladder IS

        s <= x XOR y XOR cin;

        cout <= (x AND y) OR (x AND cin) OR (y and cin);

END functlogica;
```

IMPLEMENTARE COMPORTAMENTALĂ



- arhitectura comportamentală - descrie algoritmul realizat de un modul
- instrucțiuni de tip *process* - se execută în paralel
- procesele includ instrucțiuni *secvențiale*
- procesele includ instrucțiuni *de asignare*
- procesele includ instrucțiuni *wait*

```
ENTITY reg4 IS

    port (d0, d1, d2, d3, en, clk: in BIT;

          q0, q1, q2, q3: out BIT);

END reg4
```

```
ARCHITECTURE behav OF reg4 IS

begin

        memorare:process

                variable st_d0, st_d1, st_d3 : bit;

                begin

                        if en='1' and clk='1' then

                        st_d0:=d0; st_d1:=d1; st_d2=d2; st_d3=d3;

                        end if;

                        q0 <= st_d0 after 5 ns; q1 <= st_d1 after 5 ns; q2 <= st_d2 after 5 ns; q3 <= st_d3 after 5 ns;

                        wait on d0, d1, d2, d3, en, clk;

                        end process memorare;

    end behav
```

O ALTĂ IMPLEMENTARE

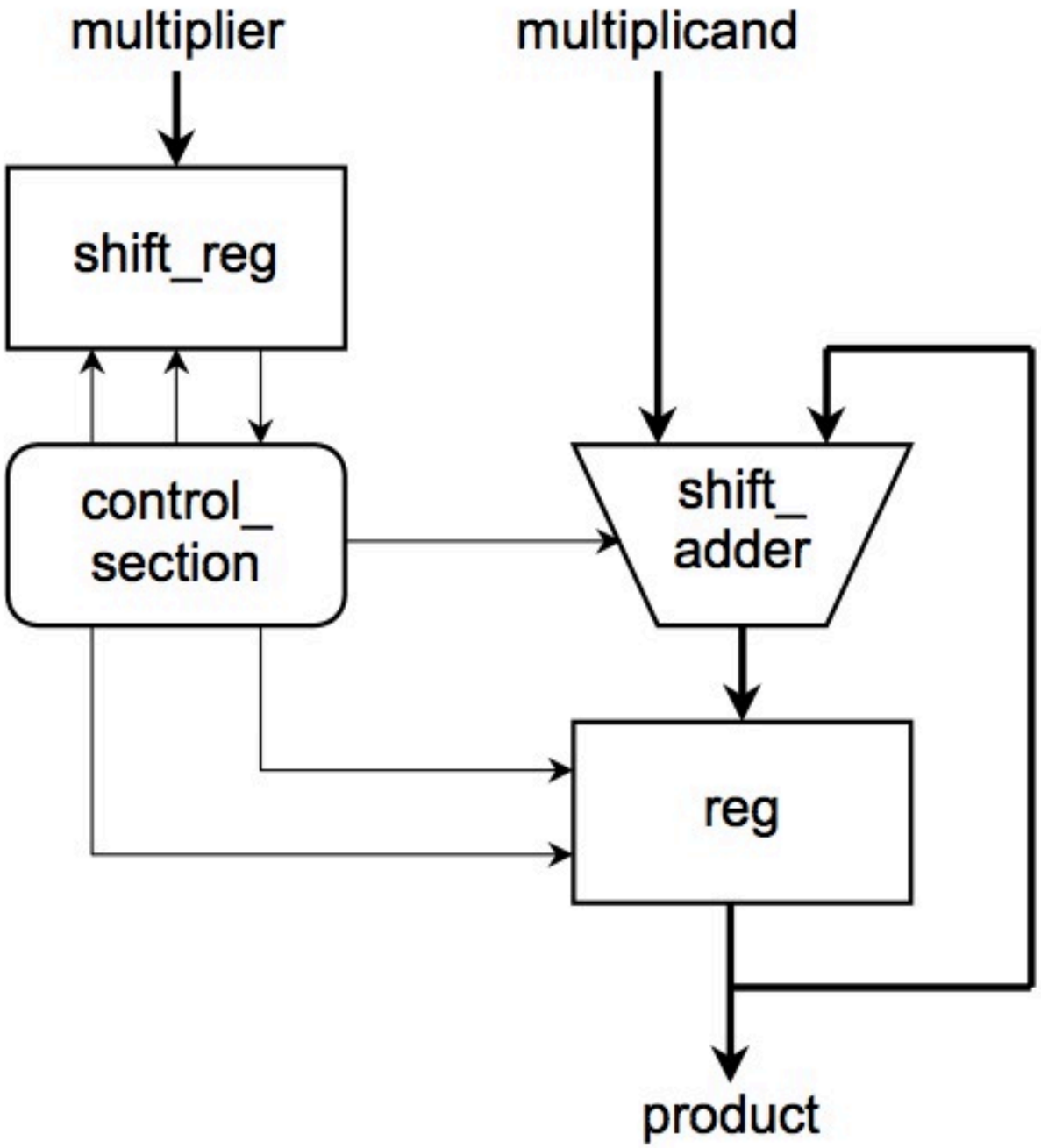
```
entity d_latch is

        port (d, clk:in bit; q:out bit);

end d_latch;

architecture basic of d_latch is

begin

        latch_behav:process

        begin

                if clk='1' then q <= d after 2 ns;

                end if;

                wait on clk, d;

        end process latch_behav

end basic;
```

```
entity and2 is

port (a, b : in bit; y : out bit);

end and2;

architecture basic of and2 is

begin

  and2_behav:process

  begin

        y <= a and b after 2 ns;

        wait on a, b;

end process and2_behav

end basic;
```

```
architecture struct of reg4 is

component d_latch port(d,clk:inbit; q:outbit);
end component;

component and2 port ( a, b : in bit;    y : out
bit );
end component; signal int_clk : bit;

.....
```

```
...
begin
    bit0 : d_latch port map ( d0, int_clk, q0 );
    bit1 : d_latch port map ( d1, int_clk, q1 );
    bit2 : d_latch port map ( d2, int_clk, q2 );
    bit3 : d_latch port map ( d3, int_clk, q3 );
    gate : and2 port map ( en, clk, int_clk );
end struct;
```

Programare combinată - structurală și comportamentală



- Calea de date o vom descrie structural
- Controlul va fi descris comportamental

```
entity multiplier is
	port ( clk, reset : in bit;multiplicand, multiplier : in integer; product : out integer );
end entity multiplier;

architecture mixed of mulitplier is
	signal partial_product, full_product : integer;
	signal arith_control, result_en, mult_bit, mult_load : bit;

begin
	arith_unit : entity work.shift_adder(behavior)

	port map ( addend => multiplicand, augend => full_product, sum => partial_product, add_control => arith_control );
	result : entity work.reg(behavior)

	port map ( d => partial_product, q => full_product, en => result_en, reset => reset );
```

```
...
multiplier_sr : entity work.shift_reg(behavior)
	port map ( d => multiplier, q => mult_bit, load => mult_load, clk => clk );
product <= full_product;

	control_section : process is
		-- declararea de variabile pentru controlul secțiunii ...
			begin

			-- instrucțiuni secvențiale pentru asignarea valorilor care vor controla semnalele --
		... wait on clk, reset;
	end process control_section;
end architecture mixed;
```

TEST BENCH

```
entity test_bench is

end entity test_bench;

architecture test_reg4 of test_bench is
      signal d0, d1, d2, d3, en, clk, q0, q1, q2, q3 : bit;

begin
dut : entity work.reg4(behav)
      port map ( d0, d1, d2, d3, en, clk, q0, q1, q2, q3 );
      stimulus : process is begin
            d0<='1'; d1<='1'; d2<='1'; d3<='1'; waitfor20ns;

            en <= '0'; clk <= '0'; wait for 20 ns;

            en <= '1'; wait for 20 ns;

            clk <= '1'; wait for 20 ns;

            d0<='0'; d1<='0'; d2<='0'; d3<='0'; waitfor20ns;

            en <= '0'; wait for 20 ns;
      ... wait;
      end process stimulus;
end architecture test_reg4;
```

Procesarea proiectului - ANALIZA

- Verificarea sintaxei și a erorilor semantice

- Analiza fiecărei unități de proiectate separat
  - declararea entității
  - corpul arhitecturii etc

- Fiecare modul al proiectării este INDICAT să fie într-un fișier separat

- Analiza modulelor care compun proiectarea sunt plasate într-o librărie - **work** este librăria curentă

Procesarea proiectului - ELABORAREA

- Crearea nivelurilor ierarhiei proiectării
  - crearea porturilor;
  - crearea semnalelor și a proceselor din interiorul corpului arhitecturii
  - pentru fiecare instanțiere se copiază entitatea și arhitectura
  - repetarea recursivă pentru toată ierarhia de module

- La finalul acesti faze se va obține o colecție de semnale și procese

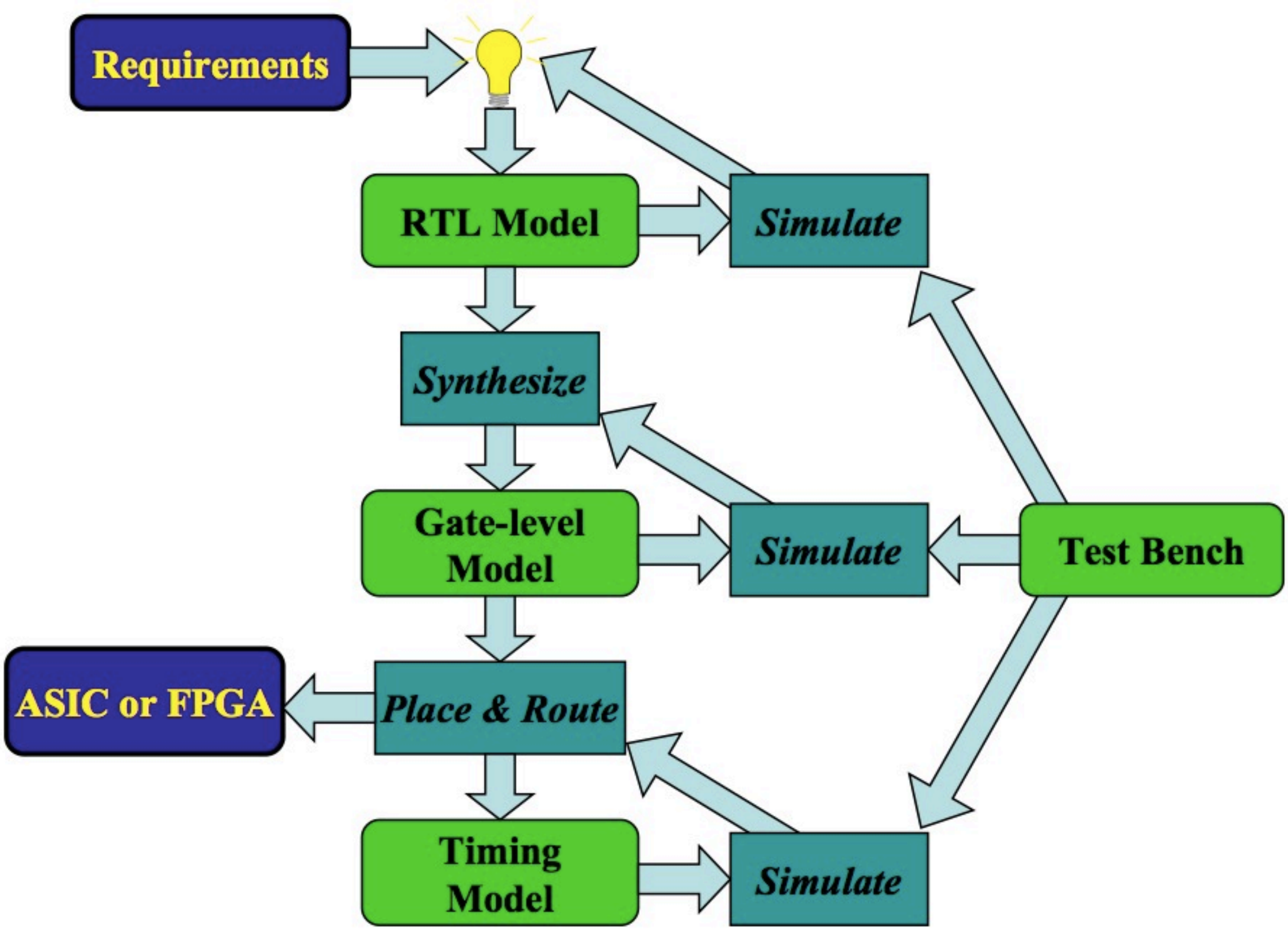
Procesarea proiectului - SIMULAREA

- Execuția proceselor din cadrul modelului elaborat
  - simularea evenimentelor discrete - mărirea timpului în pașii discreți
  - tratarea evenimentelor
  - un proces este senzitiv la evenimentele semnalelor de intrare - specificate prin intermediul instrucțiiunii WAIT
  - reluarea și programarea noilor valori ale semnalelor de ieșire
    - programarea tranzacțiilor
    - eveniment pe un semnal dacă valoarea nouă diferă de valoarea veche
  - timpul inițial de simulare este setat la 0 - fiecare semnal primește valoarea inițială
  - pentru fiecare process
    - activare
    - executarea procesului până la apariția instrucțiunii WAIT, apoi suspendare
- Simularea se încheie când nu mai există nici o tranzacție programată

	0 ns	10 ns	20 ns	30 ns
a	20	20	20	40
b	10	10	10	20
x	0	30	30	30
y	0	0	10	10
z	0	30	40	40

```
architecture behav of top is

    signal x,y,z : integer := 0;

begin
    p1 : process is
        variable a, b : integer := 0;
    begin
        a := a + 20;
        b := b + 10;
        x <= a + b after 10 ns;
        y <= a - b after 20 ns;
        wait for 30 ns;
    end process;

    p2: process is begin
        z <= (x + y);
        wait on x,y;
    end process;
end behav
```

Metodologia proiectării



Elemente lexicale

- comentarii, identificatori, simboluri speciale, numere, caractere, șiruri și șiruri de biți

- comentariile: “--“ pentru începutul unui comentariu; doar linia curentă

- identificatori: nume date de utilizator
  - trebuie să înceapă cu o literă din alfabet
  - poate conține cifre și caracterul “_”
  - nu se poate termina prin “_”
  - nu putem avea succesiuni ale caracterului “_”
- numerele: întregi sau reale
  - numerele lunigi pot fi separate: 123_456

- caracterele: trebuiesc puse între apostroafe - ‘a’, ‘A’

- șirurile de caractere: “abcde”, “12345”, “abc” & “def” => “abcdef”

- șiruri de biți
  - baza B (binară), O (octală), X (hexazecimală)
  - B”10000”, O”20”, X”10”

- simboluri speciale
  - operatorii: +, -, *, /, &
  - numerele reale
  - șirurile, șirurile de biți, delimitatori de caractere “#”
  - delimitatori lexicali: , ; :
  - specificarea precedentei: ()
  - indicii de vectori: []
  - realționali: =, >, <
  - simboluri formate din 2 caractere: :=, =>, /=, >=, **

Tipuri de date

- fiecare obiect definit are valori doar de tipul declarat

- clasele de obiecte:
  - CONSTANTE

```
constant number_of_bytes: integer := 4;

constant size, count: integer := 255;
```

  - VARIABILE

```
 ariable index, sum : integer := 0;
pc := 1;
pc:= pc + 1;
```

  - SEMNALE,
  - FIȘIERE

- tipul unei variabile determină și tipul operațiilor care se pot executa

- Avem tipuri de date predefinite, dar și tipuri de date care pot fi definite de către proiectant

- tipuri definite de utilizator

- Tipul întreg:

  varaible x, y: integer;

  type month is range 1 to 12 ;

  type count_down is range 10 downto 0;

  operații posibile: +, -, *, /, mod, rem, abs, **, =, /=, <, >, <=, >=

- Tipul real:

  varaible x, y: real ;

  type temp is range -273.0 to 1000.0;

  operații posibile: +, -, *, /, abs, **, =, /=, <, >, <=, >=

- Tipul fizic:

```
type distance is range 0 to 1E9
        units mm;
        m = 1000 mm;
        km = 1000 m;
end units distance;
```

operații posibile: +, -, * (întreg sau real), / (întreg sau real), abs, **, =, /=, <, >, <=, >=

- Tipul time - este un tip fizic predefinit

```
type time is range implementation defined units fs;
        ps = 1000 fs;
        ns = 1000 ps;
        us = 1000 ns;
        ms = 1000 us;
        sec = 1000 ms;
        min = 60 sec;
        hr = 60 min;
end units;
```

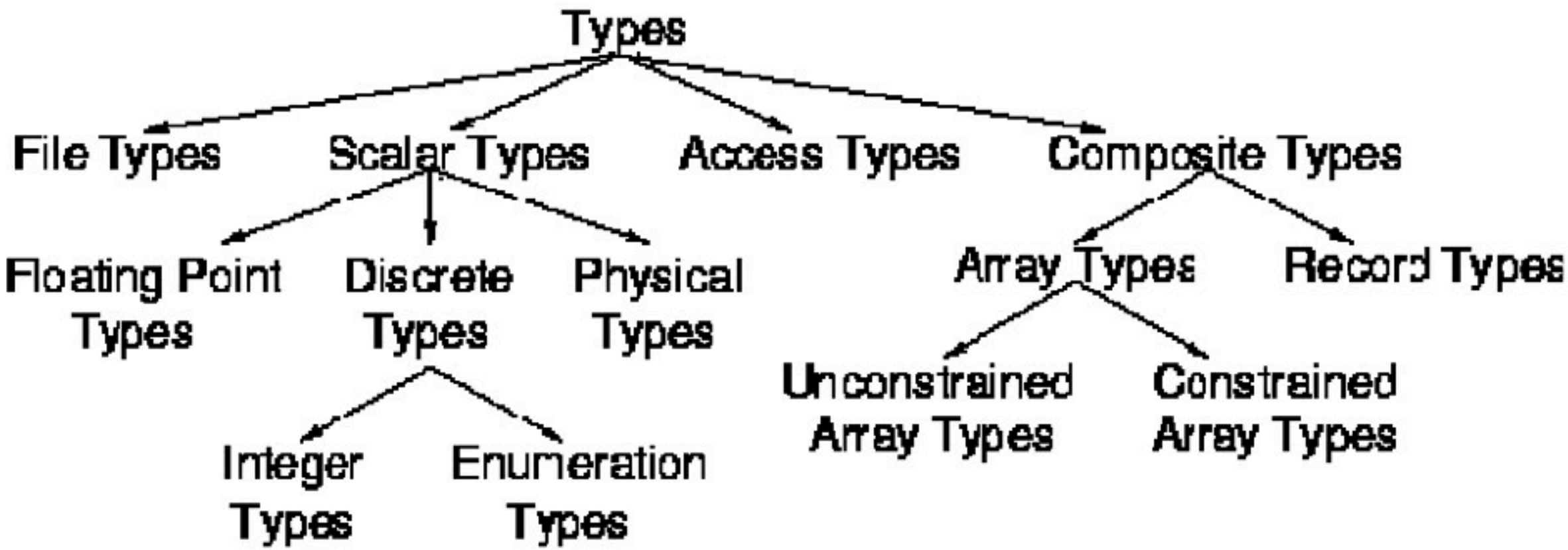
- Tipul enumerare: folosit pentru a da nume unor valori ale unnui obiect

```
type alu_func is (disable, pass, add, sub, mult, div);units mm;
```

Predefinite: caracter (=, /=, <, >, <=, >=); boolean (=, /=, <, >, <=, >=); bit (=, /=, <, >, <=, >=, and, or, nand, nor, xor, xnor, not, sll, srl, sla, sra, rol, ror)

- Subtipuri: folosite pentru limitarea rangului unui tip

```
type month is 1 to 31;
subtype working_day is 1 to 3;
        variable x,y : month;
        variable z : working_day;

        y = x + z;
```



Source:http://www.ececs.uc.edu/~paw/tyvis/doc/node7.html

Precedența operatorilor

	Operator Class	Operator
Highest precedence	Miscellaneous	**, ABS, NOT
	Multiplying	*, /, MOD, REM
	Sign	+, −
	Adding	+, −, &
	Shift	SLL, SRL, SLA, SRA, ROL, ROR
	Relational	=, /=, <, <=, >, >=
Lowest precedence	Logical	AND, OR, NAND, NOR, XOR, XNOR

Instrucțiuni secvențiale

- Instrucțiunile care apar în interiorul unui proces
- asignări de variabile
- case
- loop: infinit, while, for
- asignări de semnale
- chemarea de funcții și proceduri

Instrucțiunea IF

```
if sel = 0 then

        result <= input_0; -- executată dacă sel = 0

else

        result <= input_1; -- executată dacă sel /= 0 end if;
```

```
if sel = 0 then
        result <= input_0; -- executată dacă sel = 0
elseif sel = 1 then
        result <= input_1; -- executată dacă sel = 1
else
        result <= input_2; -- executată dacă sel /= 0, 1
end if;
```

Instrucțiunea CASE

Presupunem că dorim modelarea unei unități aritmetice logice a cărei intrare de control *func* este definită astfel:

type alu_func is (pass1, pass2, add, sub);

```
case func is
        when pass1 => results := operand1;
        when pass2 => results := operand2;
        when add => results := operand1 + operand2;
        when sub => results := operand1 - operand2;
end case;
```

Instrucțiunea NULL

Folosită în cazurile în care nu este necesară realizarea nici unei acțiuni

```
case func is
        when pass1 => results := operand1;
        when pass2 => results := operand2;
        when add => results := operand1 + operand2;
        when sub => results := operand1 - operand2;
        when nop => null;
end case;
```

Instrucțiunea LOOP

De regulă se folosește în corpul unui proces împreună cu o instrucțiune WAIT

```
p1: process is begin
        ..... L1: loop
                ..... L2 : loop
                        ..... -- cicluri imbricate .....
                end loop;
        ..... end loop;
        wait;
end process;
```

Instrucțiunea EXIT

```
loop
        ... exit ; -- controlul este în afara ciclului loop ...
end loop;
.....-- Ieșirea cauzează execuția instrucțiunilor din
acest punct
```

```
exit loop1; -- controlul în afara ciclului loop1
exit when x = 1; -- controlul în afara ciclului dacă
condiția este adevărată
```

Instrucțiunea WHILE

```
entity cos is
port (theta: in real; result: out real;);
end entity;

architecture series of cos is begin
        P1: process (theta) is
                variable sum, term, n: real;
        begin
                sum := 1.0; term := 1.0; n := 0.0;
                while abs term > abs (sum/1.0E6) loop
                        n := n + 2.0; term := (-term) * (theta ** 2) / ( (n-1) * n); sum := sum + term;
                end loop;
        result <= sum;
        end process;
end architecture;
```

Instrucțiunea FOR

```
for count in 0 to 127 loop
        count_out <= count;
        wait for 5 ns;
end loop;
```

Tipul parametrilor unui ciclu FOR este un tip de bază cu rang discret

Parametrul ciclului este constant în interiorul ciclului

Parametrul ciclului este constant în interiorul ciclului

Parametrul ciclului nu necesită o declarare prealabilă

Scopul parametrului unui ciclu FOR este limitat la acel ciclu

Nu putem avea un alt parametru cu același nume în interiorul ciclului

Instrucțiunile ASSERT și REPORT

Anumite condiții trebuiesc îndeplinite pentru funcționarea corectă a unui modul

Aceste condiții pot fi definite prin intermediul instrucțiunilor “assert”

Instrucțiunea REPORT este folosită pentru a afla informații suplimentare de la instrucțiuni de tipul assert

Aceste instrucțiuni sunt folosite în faza de debug

```
assert value <= max_value;

assert value <= max_value report “valoare prea mare”;

type severity_level is (note,warning,error,failure);

assert clock_width >= 100 ns report “lățimea ceasului prea mică” severity failure;
```

Tipuri de date composite

Vectorii

type word is array (0 to 31) of bits;
type word is array (31 to 0) of bits;

type controller_state is (initial, idle, active, error)
type state_counts is array (idle to error) of natural;

type matrix is array (1 to 3, 1 to 3) of real;
variable transform: matrix;

variable short_sample_buf: sample (0 to 63); -- indică că indexul este un număr natural de la 0 până 63

Șirurile, vectorii de biți și vectorii standard logic sunt de tipul vector fără constrângeri

Record - sunt tipuri de date care conțin valori ale unor elemente ce pot avea tipuri diferite

type time_stamp is record
seconds: integer range 0 to 59 minutes:
integer range 0 to 59 hours: integer range 0 to 23
end record time_stamp;

variable sample_time, current_time: time_stamp;

PROCESE ȘI SEMNALE

Instrucțiunile în interiorul unei arhitecturi sunt concurente - procesele

Semnalele sunt utilizate pentru a comunica între procese executate concurent

Un proces care scrie într-un semnal este denumit driver-ul acelui semnal. Două procese nu pot scrie același semnal

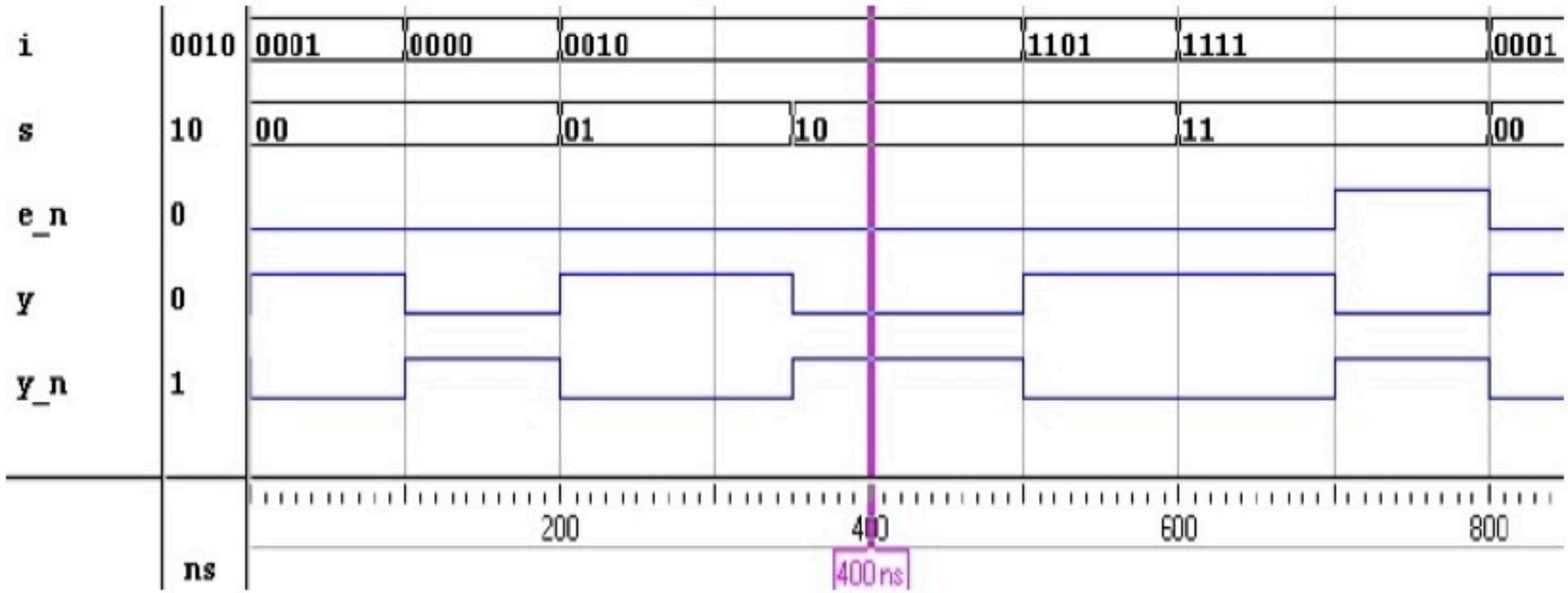
O asignare a unui semnal nu este executată imediat, este programată spre execuție în momentul în care există tranzacție petru semnal

Tranziția efectivă este realizată atunci când se execută instrucțiunea WAIT din cadrul unui proces

După ce toate procesele sunt suspendate, tranzacțiile sunt executate

---

```
E_N <= '0'; S <= "00"; I <= "0001"; wait for 100ns;
E_N <= '0'; S <= "00"; I <= "0000"; wait for 100ns;
E_N <= '0'; S <= "01"; I <= "0010"; wait for 150ns;
E_N <= '0'; S <= "10"; I <= "0010"; wait for 150ns;
E_N <= '0'; S <= "10"; I <= "1101"; wait for 100ns;
E_N <= '0'; S <= "11"; I <= "1111"; wait for 100ns;
E_N <= '1'; S <= "11"; I <= "1111"; wait for 100ns;
```



source: http://users.etech.haw-hamburg.de/users/reichardt/chapt_9.pdf

Instrucțiunea WAIT

wait for 10 ns;

wait on

wait until <=> wait on s1, s2, s3 until *condition*;

```
half_adder: process is
begin
        s <= a xor b after 10 ns;
        c <= a and b after 10 ns;
        wait on a, b;
end process;

half_adder: process (a, b) is
begin
        s <= a xor b after 10 ns;
        c <= a and b after 10 ns;
end process;
```

PROCEDURILE

```
procedure average_samples is
        variable total: real := 0.0;
                --------
                --------
end procedure average_samples;
```

Procedura poate fi invocată în interiorul unui proces astfel: average_samples;

FUNCȚIILE

Parametrii unei funcții trebuie să fie de modul “in” și nu pot fi clase variabile

```
function bv_add (bv1, bv2 : in bit_vector) return bit_vector is
        begin
                ---------------
        end function bv_add;

signal source1, source2, sum: bit_vector (0 to 31);

adder: sum <= bv_add(source1, source2) after T_delay_adder;
```

PACHETELE

O metodă importantă de organizare a datelor

O colecție de declarații grupate care servesc unui scop comun

Vizibilitatea externă a pachetului este realizată prin declararea pachetului

Implementarea pachetului este realizată prin corpul pachetului

Pachetele pot fi utilizate în mai multe modele

Există pachete predefinite precum pachetul IEEE

```
package PKG is -- declararea pachetului
        type T1 is ...
        type T2 is ...
        constant C : integer;
        procedure P1 (...);
end PKG;

package body PKG is -- corpul pachetului
        type T3 is ...
        C := 17;
        procedure P1 (...) is ... end P1;
        procedure P2 (...) is ... end P2;
end PKG;
```