CURS 4
LIMBAJUL VHDL – Partea 3
Instructiuni secventiale. Domeniul concurential. Module de simulare

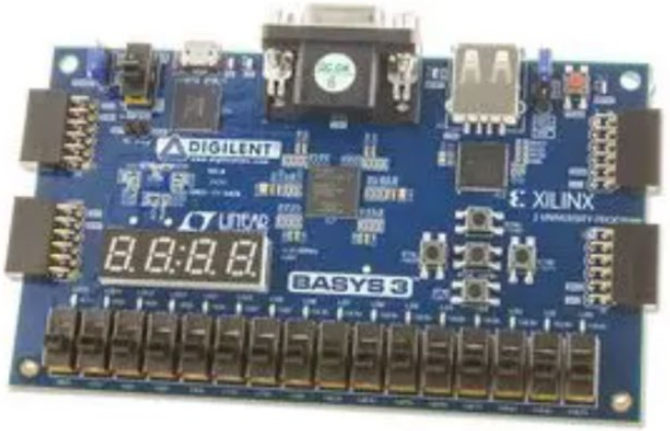
S.l. Ing. Vlad-Cristian Miclea

Universitatea Tehnica din Cluj-Napoca

Departamentul Calculatoare

Saptamanile trecute

- FPGA
- Structura unui FPGA
- Limbajul VHDL
  - Generalitati
  - Structura unui program VHDL
- Obiecte in VHDL
  - Semnale
  - Tipuri de date
  - Operatori
- Domeniul secvential
  - Procese
  - Instructiunea wait
  - Lista de sensibilitate
  - Asignarea semnalelor in procese
  - Variabile




```
a <= not b after 10 ns;
```

a	0	1	
	now	10	

```
ARCHITECTURE behavioral OF clock_component IS
BEGIN
  PROCESS
    VARIABLE periodic: BIT := '1';
      BEGIN
        IF en = '1' THEN
          periodic := not periodic;
        END IF;
        ck <= periodic;
        WAIT FOR 1 us;
  END PROCESS;
END behavioral;
```

CUPRINS

1) Introducere
2) Domeniul secvential
   - Instructiuni secventiale
   - Subprograme
3) Domeniul concurent
   - Arhitecturi concurente
   - Instructiuni concurente
4) Module de simulare
5) Concluzii

DOMENIUL SECVENŢIAL

- Process – permite instructiuni secventiale
- Se executa in bucla infinita, nu se ia nicio instructiune in considerare
- Procesul trebuie suspendat – instructiunea **WAIT**
- Conditii de revenire: lista de sensibilitate
- Semnale in procese: se fac atribuiri doar la suspendare
- Nevoie de variabile!

INSTRUCŢIUNI SECVENŢIALE

Instrucţiunea assert

- supraveghează o condiţie şi dacă este falsă emite un mesaj
- sintaxa:

assert condiţie{**report** mesaj}{**severity** nivel_de_severitate_al_erorii};

- mesajul implicit: Assertion Violation
- nivelul de severitate al erorii este de tipul Severity_Level (Note, Warning, Error, Failure), cu Error valoare implicită

INSTRUCŢIUNI SECVENŢIALE

Instrucţiunea report

- permite afişarea unui mesaj
- **sintaxa:**

{etichetă:} **report** mesaj
{nivel_de_severitate_al_mesajului};

INSTRUCŢIUNI SECVENŢIALE

Instrucţiunea de asignare a variabilelor

- valoarea este preluată imediat
- asignarea se poate face la declarare

Instrucţiunea de apel de procedură

- apelarea trebuie să indice numele şi în paranteză lista parametrilor de apel

INSTRUCŢIUNI SECVENŢIALE

Structura condiţională

- este structurată
- permite **executarea condiţionată** a unor secvenţe de instrucţiuni
- când condiţia booleană este True se execută ramura **if**, pentru False se execută ramura **else**
- ramura **elsif** permite înlănţuirea condiţiilor

INSTRUCŢIUNI SECVENŢIALE

Structura condiţională

- sintaxa:

```
if condiţie_booleană_1 then
        -- Secvenţa_de_instrucţiuni_1
elsif condiţie_booleană_2 then
        -- Secvenţa_de_instrucţiuni_2
else
        -- Secvenţa_de_instrucţiuni_3
end if;
```

INSTRUCŢIUNI SECVENŢIALE

Instrucţiunea case

- permite selectarea, în funcţie de valoarea unei expresii, a unei secvenţe de instrucţiuni dintre mai multe alternative
- expresia şi valorile trebuie să fie de acelaşi tip discret (enumerat)
- ordinea ramurilor nu contează
- ramura **others** trebuie să fie ultima şi este obligatorie dacă nu sunt specificate toate valorile posibile ale expresiei

INSTRUCŢIUNI SECVENŢIALE

Instrucţiunea case

- sintaxa:

```
case expresie is
    when Valoare_1 =>...            -- Secv_instrucţiuni_1
  when Valoare_2|Valoare_3|Valoare_4 =>          ...
                                    -- Secv_instrucţiuni_2
  when Valoare_5 to Valoare_6        =>...
                                    -- Secv_instrucţiuni_3
  ...
  when others =>...                 -- Secv_instrucţiuni_n
endcase;
```

INSTRUCŢIUNI SECVENŢIALE

Structura de buclă

- permite repetarea secvenţei de instrucţiuni din cadrul ei
- fiecare trecere se numeşte iteraţie
- dacă schema de iteraţie nu este precizată → număr infinit de iteraţii
- dacă schema este precizată → arată numărul de repetări

INSTRUCŢIUNI SECVENŢIALE

Structura de buclă

- sintaxa generală:

{etichetă:} {schemă de iteraţie} **loop**

-- Secvenţa_de_instrucţiuni

end loop {etichetă};

INSTRUCŢIUNI SECVENŢIALE

Structura de buclă

- prima schemă de iteraţie:
  - atât timp cât condiţia este adevărată se repetă instrucţiunile din secvenţă
  - testarea condiţiei - la începutul fiecărei iteraţii

```
while condiţie loop
        -- Secvenţa_de_instrucţiuni
end loop;
```

INSTRUCŢIUNI SECVENŢIALE

Structura de buclă

- a doua schemă de iteraţie:
  - secvenţa de instrucţiuni se repetă de un număr de ori cunoscut doar la execuţie
  - variabila de buclă:
    - contorizează numărul de cicluri efectuate prin parcurgerea unui tip enumerat
    - nu trebuie declarată
    - este cunoscută numai în interiorul buclei
    - nu i se poate atribui nici o altă valoare → nu poate fi modificată

INSTRUCŢIUNI SECVENŢIALE

Structura de buclă

- a doua schemă de iteraţie:
  - nu există posibilitatea specificării unui pas
  - secvenţa nu se execută pentru interval vid sau negativ

```
for Indice in 1 to 100 loop
        -- Secvenţa de instrucţiuni
end loop;
```

INSTRUCTIUNI SECVENTIALE – FOR LOOP

Exemplu secvential 1 – For loop

```
p_Shift_With_For : process (i_Clock)
  begin
    if rising_edge(i_Clock) then
      for ii in 0 to 2 loop
        r_Shift_With_For(ii+1) <=
r_Shift_With_For(ii);
      end loop;  -- ii
    end if;
  end process;
```

Exemplu secvential 2 - Atribuiri

```
p_Shift_Without_For : process (i_Clock)
  begin
    if rising_edge(i_Clock) then
      r_Shift_Regular(1) <= r_Shift_Regular(0);
      r_Shift_Regular(2) <= r_Shift_Regular(1);
      r_Shift_Regular(3) <= r_Shift_Regular(2);
    end if;
  end process;
```

INSTRUCŢIUNI SECVENŢIALE

Instrucţiunea next

- permite oprirea iteraţiei în curs de desfăşurare a unei bucle
- execuţia continuă cu iteraţia următoare (dacă există)
- poate fi:
  - imperativă: **next** {eticheta_buclei}**;**
  - condiţională (întrerupere când condiţia este adevărată): **next** {eticheta_buclei} **when** condiţie**;**
- fără etichetă se referă la bucla cea mai de jos

INSTRUCŢIUNI SECVENŢIALE

Instrucţiunea exit

- permite ieşirea din buclă
- întrerupe toate iteraţiile restante ale buclei
- execuţia continuă cu instrucţiunea de după **end loop**
- poate fi:
  - imperativă: **exit** {eticheta_buclei};
  - condiţională (ieşire când condiţia este adevărată): **exit** {eticheta_buclei} **when** condiţie;
- fără etichetă se referă la bucla cea mai de jos

INSTRUCŢIUNI SECVENŢIALE

Instrucţiunea return

- este rezervată subprogramelor
- la execuţia ei se suspendă subprogramul şi controlul revine apelantului
- poate fi folosită pentru a întrerupe o procedură şi a reveni în programul apelant → nu trebuie să i se asocieze o valoare: **return**;

INSTRUCŢIUNI SECVENŢIALE

Instrucţiunea return

- orice funcţie se termină dinamic prin **return** urmată de valoarea returnată:

  return Valoare;

- valoarea returnată trebuie să aibă tipul declarat în specificaţia funcţiei
- o funcţie poate avea mai multe instrucţiuni **return**, pentru că pot exista mai multe ramuri de decizie

INSTRUCŢIUNI SECVENŢIALE

Instrucţiunea nulă

- se trece la executarea instrucţiunii următoare
- sintaxa: **null**;
- practic se foloseşte la instrucţiuni de selecţie (case) când toate ramurile trebuie luate în considerare
- nu este necesară la compilarea unui proces sau a unui corp de procedură vid

SUBPROGRAME

Generalităţi

- permit scrierea unor algoritmi reutilizabili
- valorile parametrilor, diferite la apel, duc la efecte diferite
- 2 tipuri de subprograme:
  - proceduri: **procedure**
  - funcţii: **function**
- apelul unei proceduri este o instrucţiune
- apelul unei funcţii apare ca o valoare, în membrul drept al instrucţiunilor de asignare

SUBPROGRAME

Declaraţia de subprogram

- subprogramele au 2 părţi:
  - declaraţia (specificaţia)
  - corpul
- declaraţia indică:
  - genul subprogramului (procedură sau funcţie)
  - numele
  - lista parametrilor formali (fiecare cu mod şi tip)
  - pt. funcţie şi tipul valorii returnate

SUBPROGRAME

Declaraţia de subprogram

- sintaxa:
  - procedură

procedure nume_procedură (lista_parametrilor_formali);

  - funcţie

{**pure/impure**} **function** nume_funcţie
(lista_parametrilor_formali)

return tipul_rezultatului;

  - lista parametrilor formali

{clasă_obiect} nume_parametru_1{, nume_parametru_2}:

{mod_transmitere} **type** valoare_implicită;

SUBPROGRAME

Modul de transmitere a parametrilor

- intrare - **in**
  - implicit
  - parametrii pot fi citiţi, nu pot fi modificaţi
- ieşire - **out**
  - numai pentru proceduri
  - parametrii nu se pot citi
- combinat intrare / ieşire - **inout**
  - numai pentru proceduri
  - permite orice citire şi scriere

SUBPROGRAME

Corpul subprogramului

- conţine algoritmul implementat
- nu permite declararea semnalelor
- sintaxa:

antet_sub-program **is**

{partea declarativă}

begin

{partea rezervată instrucţiunilor}

end {nume_sub-program};

SUBPROGRAME

Apelul

- poate fi secvenţial sau concurent
- 2 moduri de indicare a parametrilor actuali ai subprogramelor:
  - prin poziţie
  - prin denumire (cu =>)

SUBPROGRAME

Supraîncărcarea

- 2 subprograme cu acelaşi nume, dar profiluri diferite
- profilul: numărul, ordinea şi tipul parametrilor formali şi tipul rezultatului pentru funcţie

EXEMPLU DESCRIERE

Decrieti un numarator

- Circuit secvential, care incrementeaza vechea valoare la fiecare ciclu de ceas
  - Poate fi resetat
  - Poate avea un enable
- Ce fel de descriere se poate folosi?
- De ce fel de structuri VHDL avem nevoie?

ARHITECTURI CONCURENTE

Generalităţi

- VHDL descrie sistemele ca mulţimi de subsisteme funcţionale care operează în mod concurent
- orice subsistem poate fi specificat printr-un proces separat → nivel de detaliere dat de necesităţi
- toate procesele din interiorul unei arhitecturi se execută concurent

⇒ în VHDL → combinare între operaţii concurente şi secventiale

ARHITECTURI CONCURENTE

Generalităţi

- transferul de informaţii între procese se face prin semnale
- procesele se activează indiferent dacă modificarea valorii semnalelor la care sunt active este produsă de mediul extern sau de către alt proces

ARHITECTURI CONCURENTE

“Procese" elementare

- procesele care conţin o singură instrucţiune
  → instrucţiuni singulare de asignare concurentă de semnal
- instrucţiunile singulare de asignare concurentă de semnal:
  - apar în arhitecturi şi se execută concurent cu alte procese
  - sunt sensibile la modificarea oricărui semnal care apare în membrul drept
  - asignarea se poate întârzia cu **after**
- **Simplu in cazul in care nu exista “concurenta”**

ARHITECTURI CONCURENTE

Procese elementare - exemple

```
entity Demux1la4 is
port (i: in bit;                                    -- intrare
      s: in bit_vector (1 downto 0);                -- selecţii
      d: out bit_vector (3 downto 0));              -- ieşiri
end Demux1la4;
architecture Conc of Demux1la4 is
signal t : bit_vector(3 downto 0);                  -- semnal intern
begin
t(3)<=s(1) and s(0);                                -- selectarea unei ieşiri
t(2)<=s(1) and not s(0);
t(1)<=not s(1) and s(0);
t(0)<=not s(1) and not s(0);
d<=i and t;                                         -- valoarea ieşirii
end Conc;
```

ARHITECTURI CONCURENTE

Valori de semnale - pilotul (driver)

- semnalele primesc valori noi în momentul suspendării proceselor prin instrucţiunea **wait**
- stocarea informaţiilor referitoare la evenimentele de pe semnal → prin pilot (driver)
- compilatorul creează un pilot pentru fiecare semnal care primeşte o valoare într-un proces
- există un singur pilot / semnal / proces

ARHITECTURI CONCURENTE

Valori de semnale - pilotul (driver)

- toate operaţiile sunt efectuate asupra pilotului
- pilotul copiat în semnal când procesul se suspendă
- semnalul cunoaşte valorile trecute, prezente şi viitoare → pilotul are asignată o formă de undă
- forma de undă formată din tranzacţii
- tranzacţia: pereche valoare semnal + valoare timp (Time)

ARHITECTURI CONCURENTE

Semnale multi-sursa (cu mai mulţi piloţi)

- problema principala in domeniul concurential
- Semnale care au informaţii provenite din **mai multe surse**
  - exemplu: magistrala internă a procesorului primeşte informaţii de la: procesor, memoria internă, hard discuri, dispozitive de intrare / ieşire
- necesită stabilirea unei metode de determinare a valorii rezultate pentru semnal

ARHITECTURI CONCURENTE

Rezolvarea semnalelor multi-sursă

- simulatorul nu poate şti dacă un semnal va fi activat din mai multe surse
- simulatorul trebuie să fie pregătit pentru a realiza “rezolvarea” (mixarea) semnalelor
- regulile de mixare se specifică într-un **tabel** care reprezintă o funcţie de rezoluţie → conţine toate valorile posibile pentru semnal

ARHITECTURI CONCURENTE

Funcţia de rezoluţie

- semnalele rezolvate - au funcţie de rezoluţie
- funcţia de rezoluţie poate să apară la declaraţii de subtipuri de date şi la declaraţii de semnal
- la tipurile compuse funcţia de rezoluţie pentru tipul compus maschează funcţiile de rezoluţie pentru elementele tipului compus
- în simulare funcţia de rezoluţie este utilizată automat, nu este controlată de proiectant

ARHITECTURI CONCURENTE

Funcţia de rezoluţie

- Bit nu permite semnale multi-sursa; Std_logic este preferabil
- **Std_Logic:** tip de date rezolvat din pachetul **Std_Logic_1164**
- există şi versiunea rezolvată pentru **Std_Logic_Vector**

INSTRUCŢIUNI CONCURENTE

Paralelism

- la instrucţiunile concurente ordinea de execuţie este oarecare
- Insructiunile concurente ruleaza in acelasi timp
- Se pot folosi mai multe (oricate, atat timp cat nu se depasesc resursele) componente in acelasi timp => foarte util pentru paralelism

INSTRUCŢIUNI CONCURENTE

Instrucţiunea block

- are 3 funcţii principale:
  - încapsularea declaraţiilor - reunire de instrucţiuni concurente care au acces la declaraţii locale
  - utilizarea instrucţiunilor gardate - permite scrierea de instrucţiuni de asignare condiţionate
  - suport pentru ierarhizare - se pot scrie proiecte ierarhizate
- blocul este unitatea de bază echivalentă a structurării în VHDL

INSTRUCŢIUNI CONCURENTE

Instrucţiunea block

- sintaxa:

etichetă: **block** {(condiţie_de_gardă)}
{antet_generice_şi_porturi}
-- Declaraţii locale
begin
-- Instrucţiuni concurente
end block {etichetă};

INSTRUCŢIUNI CONCURENTE

Instrucţiunea block

- condiţia de gardă = expresie booleană
- antetul opţional indică importarea din mediul exterior:
  - valori pentru parametri generici
  - semnale pentru porturi
  - partea declarativă:
    - vizibilă numai local
  - nu permite declaraţii de variabile locale

INSTRUCŢIUNI CONCURENTE

Apel concurent de procedură

- aceeaşi sintaxă ca la apelul secvenţial
- are în plus în procesul echivalent instrucţiuni **wait**
- parametrii pot fi doar constante sau semnale
- util la aplicaţii de gestionare a stării interne a automatelor finite

INSTRUCŢIUNI CONCURENTE

Instrucţiunea assert

- aceeaşi sintaxă ca la apelul secvenţial
- poate să apară în entităţi sau arhitecturi
- monitorizarea condiţiei date este permanentă

INSTRUCŢIUNI CONCURENTE

Instrucţiunea de asignare de semnal

- are 2 forme: condiţională şi selectivă
- sintaxa formei condiţionale:

{etichetă:} nume_sau_agregat <= {**guarded**}

formă_de_undă_1 **when** condiţie_booleană_1 **else**

formă_de_undă_2 **when** condiţie_booleană_2 **else**

…

formă_de_undă_n;

INSTRUCŢIUNI CONCURENTE

Instrucţiunea de asignare de semnal

- forma condiţională echivalentă (procesul echivalent):

```
if condiţie_booleană_1 then
    nume_sau_agregat <= {transport} formă_de_undă_1;
elsif condiţie_booleană_2 then
    nume_sau_agregat <= {transport} formă_de_undă_2;
...
else
    nume_sau_agregat <= {transport} formă_de_undă_n;
end if;
```

INSTRUCŢIUNI CONCURENTE

Instrucţiunea de asignare de semnal

- sintaxa formei selective:

{etichetă:} **with** expresie **select**

nume_sau_agregat <= {**guarded**}{**transport**}

formă_de_undă_1 **when** alegere_1,

formă_de_undă_2 **when** alegere_2,

…

formă_de_undă_n **when** alegere_n;

INSTRUCŢIUNI CONCURENTE

Instrucţiunea de asignare de semnal

- forma selectivă echivalentă (procesul echivalent):

```
case expresie is  when
alegere_1 =>
    nume_sau_agregat <= {transport} formă_de_undă_1;
when alegere_2 =>
    nume_sau_agregat <= {transport} formă_de_undă_2;
        …
when alegere_n =>
    nume_sau_agregat <= {transport} formă_de_undă_n;
end case;
```

INSTRUCŢIUNI CONCURENTE

Instrucţiunea de instanţiere a unei componente

- ia o copie a unui model (componentă) declarat anterior şi o personalizează pentru o necesitate particulară
- sintaxa:

etichetă: numele_componentei_model

{corespondenţa parametrilor generici}

{corespondenţa porturi efective / porturi locale modelului};

INSTRUCŢIUNI CONCURENTE

Instrucţiunea de instanţiere a unei componente - exemplu

```
-- Avem două componente, C1 şi C2, care asigură
-- interconectarea semnalelor A, B, C şi D de tip Bit.
signal A, B, C, D: Bit;
-- A, B, C şi D sunt porturile efective ale circuitului proiectat
-- Declararea componentei Inversor
component: Inversor
    port(Intrare: in Bit; Ieşire: out Bit);
end component;
begin
-- Componenta Inversor va fi instanţiată de două ori:
 C1: Inversor port map (Intrare => A, Ieşire => B);
 C2: Inversor port map (Intrare => C, Ieşire => D);
```

INSTRUCŢIUNI CONCURENTE

Bloc sau componentă

- nu există o diferenţă fundamentală între un bloc şi o instanţă de componentă
- ambele pot fi folosite la descrierea ierarhizată
- componenta are avantajul că este reutilizabilă

INSTRUCŢIUNI CONCURENTE

Instrucţiunea generate

- permite elaborarea condiţională sau iterativă a liniilor de cod sursă VHDL
- forma condiţională - instrucţiunile concurente vor exista la elaborare după îndeplinirea condiţiei booleene
- forma iterativă - creează ansambluri de instrucţiuni în număr egal cu numărul de elemente din intervalul discret

INSTRUCŢIUNI CONCURENTE

Instrucţiunea generate

- sintaxa:
  - forma condiţională

etichetă: **if** condiţie_booleană **generate**

… Secvenţă de instrucţiuni concurente

end generate {etichetă};

  - forma iterativă

etichetă: **for** nume_parametru_de_generare **in** interval_discret **generate**

… Secvenţă de instrucţiuni concurente

end generate {etichetă};

MODULE DE SIMULARE

Scop

- orice proces de proiectare presupune şi etapa de verificare
- în VHDL - mai multe metode de verificare
- modul de simulare (test bench) = mediu în care un proiect ≡ UST (unitate supusă testării) este verificat prin aplicarea unor semnale numite stimuli şi observarea răspunsurilor generate

MODULE DE SIMULARE

Elemente

- soclu (socket) - în el se plasează sistemul testat
- generator de stimuli - subsistem care aplică stimuli proiectului testat
  - stimuli generaţi intern
  - stimuli preluaţi de la o sursă externă de semnale
- instrumente de monitorizare a răspunsurilor la stimuli, generate de sistemul supus testării

MODULE DE SIMULARE

Elemente în VHDL

- modulul de simulare este o specificaţie VHDL, care este simulată de simulatorul VHDL integrat în mediul de dezvoltare VHDL
- modulul de simulare este alcătuit din:
  - instanţierea unităţii supuse testării (UST)
  - procese sensibile la stimuli aplicaţi unităţii supuse testării (UST)
- **specificaţie hibridă**: structurală + comportamentală

Observaţie: UST trebuie să existe (entitate + arhitectură)

MODULE DE SIMULARE

Elemente în VHDL

- stimulii:
  - pot fi specificaţi în arhitectura modulului de simulare
  - pot fi citiţi din fişier extern

  reacţiile unităţii testate pot fi observate prin:
- reacţiile unităţii testate pot fi observate prin:
  - formele de undă generate de simulatorul VHDL
  - fişierele de raport cu mesaje generate de simulator
  - mesajele generate de simulator la consolă
  - scrierea în fişiere folosind operaţiile de intrare / ieşire în mod text disponibile în pachetul Textio

MODULE DE SIMULARE

Structură

- specificaţie cu entitate şi arhitectură
- entitatea modulului de simulare:
  - nu are porturi şi parametri generici pentru că modulul de simulare nu este un dispozitiv real
  - nu poate lipsi pentru că arhitecturile nu pot fi specificate fără entităţi asociate!!!
- arhitectură - se realizează instanţierea UST

  = o specificaţie de tip structural - relaţia dintre modulul de simulare şi UST
  - poate fi instanţiere directă
  - instanţierea unei componente

MODULE DE SIMULARE

Structură

- stimulii - element esenţial:
  - set de semnale declarate intern în arhitectura modulului de simulare
  - asignaţi porturilor UST la instanţierea ei, prin **port map**
  - definiţi ca forme de undă:
    - prin instrucţiuni concurente de asignare de valori la semnale
    - în cadrul unuia sau a mai multor procese comportamentale, cu instrucţiuni **wait for** şi la sfârşit cu o instrucţiune **wait** vidă pentru suspendarea procesului

MODULE DE SIMULARE

Structură

- arhitectura modulului de simulare fiind în domeniul concurent - nu contează ordinea în care este instanţiată UST şi sunt definiţi stimulii

MODULE DE SIMULARE

Utilizare

- la sisteme complexe simularea poate fi costisitoare
- verificare mai simplă cu module de simulare
- se simulează modulul de simulare, nu unitatea supusă testării (ea este doar o instanţă de componentă)
- nu există limitări la dimensiunea modulului de simulare

MODULE DE SIMULARE

Exemple - poartă ŞI cu 2 intrări

```
-- Entitatea modulului de simulare
entity ModulSimulare is
end ModulSimulare;
architecture ArhModulSimulare of ModulSimulare is
-- Declararea componentei
component Poarta_ŞI is
    port (A, B: in Bit; Y: out Bit);
end component;
-- Declararea stimulilor
signal A, B, C: Bit;  begin
-- Instanţierea unităţii supuse testării (UST)
UST: Poarta_ŞI port map (A, B, C);
-- Semnalele de stimulare – ca forme de undă
    A <= '0', '1' after 20 ns, '0' after 40 ns, ‘1’ after 60 ns;
    B <= '1', '0' after 40 ns, '1' after 80 ns;
end ArhModulSimulare;
```

MODULE DE SIMULARE

Exemplu - multiplexor 4:1

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity Test_Mux4la1 is
end;
architecture Testare of Test_Mux4la1 is
component Mux4la1
port (S: in Std_Logic_Vector (1 downto 0);
        I0, I1, I2, I3: in Std_Logic; Y: out Std_Logic);
end component;
signal Sel: Std_Logic_Vector (1 downto 0);
signal A, B, C, D, F: Std_Logic;
begin
    Sel <= "00", "01" after 30 ns, "10" after 60 ns, "11" after 90 ns, "XX" after
    120 ns, "00" after 130 ns;
    A <= 'X', '0' after 10 ns, '1' after 20 ns;
    B <= 'X', '0' after 40 ns, '1' after 50 ns;
    C <= 'X', '0' after 70 ns, '1' after 80 ns;
    D <= 'X', '0' after 100 ns, '1' after 110 ns;  UST:
Mux4la1 port map (Sel, A, B, C, D, F);
    end Testare;
```

MODULE DE SIMULARE

Afişare şi raportare rezultate

- verificarea trebuie să şi afişeze sau să raporteze rezultatele
- modalităţi de afişare şi raportare:
  - afişarea listei valorilor semnalelor care se modifică în timp (echivalentă cu afişarea formelor de undă)
  - scrierea rezultatelor simulării într-un fişier (log file)
  - folosirea instrucţiunii **assert**

MODULE DE SIMULARE

Instrucţiunea assert

- utilizată pentru raportarea răspunsurilor eronate generate de unitatea supusă testării
- moduri posibile de utilizare:
  - se aplică o instrucţiune **assert** de fiecare dată când se aşteaptă o nouă valoare a unui semnal de ieşire al UST
  - valoarea prognozată se specifică drept condiţie
  - se folosesc mesaje de eroare precise şi detaliate (CE nu funcţionează şi CÂND a avut loc evenimentul)

PACHETE STANDARD ŞI PREDEFINITE

Pachete standard

- Standard
- Textio
- definite în manualul de referinţă VHDL
- nu pot fi modificate de proiectanţi

PACHETE STANDARD ŞI PREDEFINITE

Pachete predefinite

- **Std_Logic_1164** - normă IEEE
- IEEE Numeric_Std
- IEEE Numeric_Bit
- **Std_Logic_Arith** al firmei Synopsys
- **Std_Logic_Unsigned** al firmei Synopsys
- ...

Concluzii

- Domeniul secvential. Procese
  - Wait, lista de sensibilitate
  - Semnale vs variabile
- Instructiuni secventiale
  - Assert, report
  - If, case, loops
  - Subprograme (domeniul secvential)
- Domeniul concurential
  - Procese elementare
  - Pilot de semnal; semnale multi-sursa
- Instructiuni concurente
  - Paralelism!
  - Bloc
  - Asignare, selectie – descriere flux de date
  - Instantiere componente – descriere structural + generate
- Module de simulare
  - Structura unui modul de simulare
  - Afisare si verificare rezultate