LUCRAREA NR. 3
SEMNALE. PARAMETRI GENERICI. CONSTANTE

1. Scopul lucrării

Lucrarea prezintă unul dintre obiectele fundamentale de proiectare în VHDL: *semnalul*. Este detaliat *conceptul de semnal*, precum şi noţiunile fundamentale referitoare la acesta (vizibilitatea semnalelor, reguli de asignare, modele de transmisie etc.). Se introduce şi se detaliază şi conceptul de *parametri generici* (sunt prezentate exemple, domeniul de aplicabilitate, avantaje şi dezavantaje) şi cel de constantă.

2. Consideraţii teoretice

2.1 Conceptul de semnal. Definiţie

Semnalul este un purtător de informaţie. În general vorbind, el este un *fenomen fizic care se poate modifica în timp şi / sau în spaţiu, iar modificările sale pot fi specificate prin instrucţiuni formale*.

În VHDL, clasificarea fundamentală a semnalelor este următoarea:

- *Semnale externe*, purtătoare de informaţie între diferitele dispozitive electronice. Ele formează interfaţa fiecărui sistem. Semnalele externe se declară numai în partea de *entitate* a sistemului descris în VHDL.
- *Semnale interne*, purtătoare de informaţie în interiorul dispozitivelor electronice. Ele nu sunt vizibile, fiind complet încapsulate în interiorul dispozitivului şi constituind o parte a arhitecturii sale interne. Semnalele interne se declară numai în partea de *arhitectură* a sistemului descris în VHDL.

Semnalele electrice joacă un rol crucial în dispozitivele electronice. Ele sunt cele mai importante obiecte din orice dispozitiv electronic. Deoarece limbajul VHDL este proiectat pentru a descrie funcţionarea unor sisteme complete, semnalele joacă un rol important în descrierea comunicării dintre circuitele sau blocurile lor.

Semnalele permit analizarea relaţiilor temporale în cadrul sistemului. Spre deosebire de variabilele din limbajele de programare clasice, cum ar fi

Pascal sau C, semnalele VHDL conţin informaţii atât despre valorile curente cât şi despre cele trecute şi viitoare. Aceste informaţii sunt numite *istoria* (*history*) *semnalului*.

Liniile de semnal pot fi implementate ca linii *singulare* sau *multiple*.

O *conexiune singulară* este reprezentată de către o singură linie de semnal, care are o singură valoare binară la un moment dat. Un exemplu de asemenea semnal este tactul care sincronizează toate blocurile sistemului.

Conexiunile multiple se mai numesc magistrale sau vectori, acestea transmiţând informaţiile ca o combinaţie de valori binare.

În VHDL, un semnal corespunde reprezentării hardware a conceptului de purtător de informaţie. Reprezentarea poate fi o structură de date simplă sau complexă, în funcţie de tipul datelor purtate de semnal. Orice structură de date este în primul rând declarată ca un semnal, cu excepţia anumitor condiţii particulare. Orice declaraţie de semnal se face în domeniul concurent, deoarece un semnal permite conectarea mai multor module hardware (ele însele fiind definite sub formă de componente sau de blocuri din domeniul concurent). Un semnal reprezintă aşadar o definiţie concurentă. Zona declaraţiilor concurente corespunde *zonei de declarare a arhitecturilor şi entităţilor*. *Asignarea de valori unui semnal* permite evoluţia în timp a valorilor asociate cu semnalul respectiv. Faptul că valoarea unui semnal poate fi utilizată în cadrul unei expresii şi asignată de către o altă instrucţiune în acelaşi timp, va impune conservarea valorii curente a semnalului separat de valorile viitoare (generate de instrucţiunea care modifică valoarea semnalului).

În plus, pentru a facilita modelarea anumitor componente, este util să putem avea acces la valorile trecute ale oricărui semnal. Prin urmare, orice semnal poate păstra o *istorie* a valorilor sale trecute, valoarea prezentă şi chiar şi valorile prevăzute pentru viitor. Memorarea valorilor se face în aşa-numitul *pilot* (*driver*) al semnalului. În realitate, valorile trecute nu sunt conservate direct în semnalul respectiv, ci într-o versiune întârziată în timp a acestuia. De fapt, se memorează acele evenimente care indică o schimbare de valoare la un moment de timp bine definit. Numai valorile (evenimentele) viitoare pot fi modificate de către operaţia de asignare a unei valori unui semnal. Valoarea prezentă nu poate fi modificată. Operaţia de asignare a unei valori unui semnal poate fi efectuată fie prin conectarea la un port de ieşire al unei componente, fie prin operaţia de asignare în domeniul concurent, fie prin operaţia de asignare în domeniul secvenţial. Operaţia de asignare a unei valori unui semnal în domeniul concurent corespunde stilului de descriere „flux de date”.

2.2 Declararea semnalelor

Manualul de referinţă al limbajului VHDL defineşte *portul* drept „*un canal de comunicare dinamică între un bloc (sau o entitate) şi mediul său înconjurător*". Prin urmare, semnalele care conectează un modul cu mediul său înconjurător sunt numite *porturi* şi sunt specificate în secţiunea specială **ports** a declaraţiei unei entităţi. Fiecare semnal care leagă modulul hardware cu mediul său exterior e specificat ca un port în interiorul declaraţiei de entitate.

Fiecare semnal trebuie să posede:

- un *nume unic* (de exemplu, *Clock* sau *Data*[0 **to** 7]);
- un *tip* (de exemplu, BIT sau BIT_VECTOR).

Fiecare semnal declarat ca port poate avea specificată o *valoare iniţială*. Pe lângă aceste atribute indispensabile fiecărui semnal, un port trebuie să aibă, în plus, şi un atribut care să ofere informaţii despre sensul fluxului de informaţie vehiculat prin acel port: acesta se numeşte *modul* portului (*mode*).

Sensul fluxului de informaţie vehiculat prin port este crucial şi trebuie specificat de fiecare dată dacă un semnal este:

- *de intrare* (*input*): **in**;
- *de ieşire* (*output*): **out**;
- *bidirecţional* (*inout*): **inout**.

Sensul fluxului de informaţie trebuie declarat în mod explicit. Dacă nu este specificat explicit, modul implicit este **in**. Prin urmare, declararea unui port se realizează respectând următoarea sintaxă:

```
nume_port: mod tip_port;
```

Deşi în VHDL există cinci tipuri de moduri posibile ale unui semnal (**in**, **inout**, **out**, **buffer** şi **linkage**) este recomandat să se utilizeze numai primele trei dintre ele, deoarece ultimele două nu sunt acceptate de toate instrumentele de sinteză. Cele mai folosite trei moduri pot fi descrise astfel:

- modul **in**: datele pot fi citite în interiorul modulului, dar nu pot fi scrise de către arhitectura internă a modulului;
- modul **out**: datele pot fi generate în interiorul arhitecturii, dar nu pot fi citite;
- modul **inout**: datele pot fi citite şi scrise în interiorul arhitecturii.

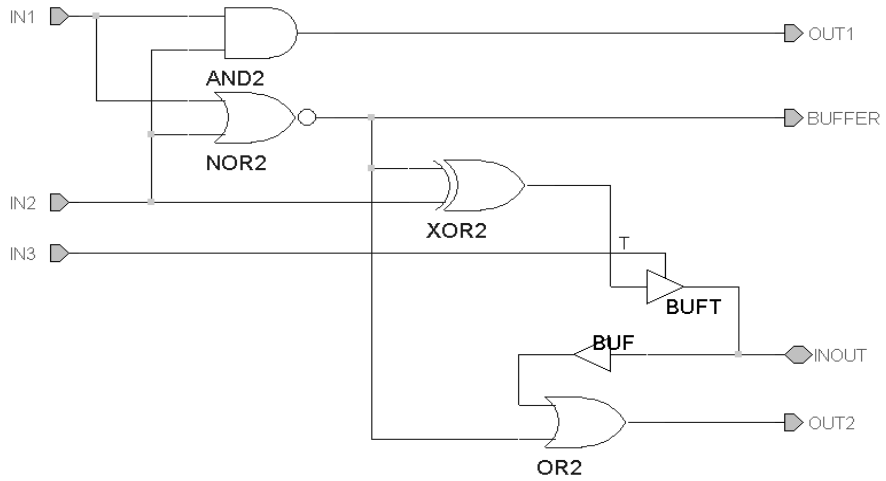

Figura 3.1 *Principalele moduri ale porturilor*

Modurile **buffer** şi **linkage** pot fi descrise astfel:

- modul **buffer**: este asimilabil modului „ieşire” şi trebuie în mod obligatoriu să nu aibă decât o singură sursă. Cu această restricţie, va fi posibilă citirea valorii vehiculate de port în interiorul arhitecturii (vezi figura 3.1);
- modul **linkage**: este mai puţin utilizat. Într-adevăr, dacă un port formal de mod **linkage** poate fi conectat la oricare alt port, porturile interne deja conectate nu mai pot fi conectate decât la alte porturi de mod **linkage**. Nu este recomandabilă utilizarea acestui mod.

În completarea acestei descrieri, recomandăm consultarea figurii 2.2 din lucrarea nr. 2.

De exemplu, pentru un modul de memorie (ROM sau RAM):

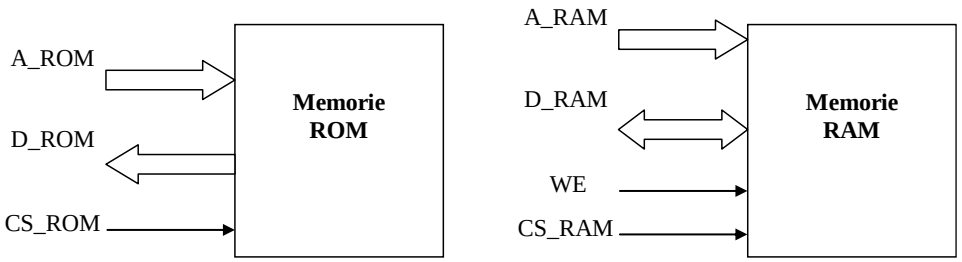

Figura 3.2 *Schema pentru memorie ROM şi pentru memorie RAM*

```
entity MEMORIE_ROM is
 port(A_ROM: in BIT_VECTOR (3 downto 0); -- Adresele
      CS_ROM: in BIT; -- Semnal selecţie cip, „Chip Select”
       D_ROM: out BIT_VECTOR(7 downto 0)); -- Ieşiri de date
end MEMORIE_ROM;
```

```
entity MEMORIE_RAM is
 port(A_RAM: in BIT_VECTOR (3 downto 0); -- Adresele
       CS_RAM: in BIT; -- Semnal selecţie cip, „Chip Select”
       WE: in BIT;      -- Semnal validare scriere, „Write Enable”
       D_RAM: inout BIT_VECTOR(7 downto 0)); -- Bidirecţional
end MEMORIE_RAM;
```

Entitatea specifică interfaţa sistemului cu mediul exterior, în timp ce *arhitectura* descrie totalitatea componentelor rezidente în interiorul sistemului. Semnalele interne nu fac excepţie de la această regulă. Pentru a putea fi deosebite de alte obiecte din codul VHDL, se utilizează cuvântul cheie **signal** pentru fiecare declaraţie.

Observaţii

1. Cuvântul cheie **signal** nu este necesar în declaraţia porturilor, deoarece fiecare port este prin definiţie un semnal.
2. Semnalele interne nu au nevoie de declaraţii de mod (**in**, **inout**, **out** etc.)
3. Există posibilitatea să se specifice o valoare iniţială pentru fiecare semnal intern.

De exemplu, pentru dispozitivul din figura 3.3, declararea semnalelor externe şi interne se face în felul următor:

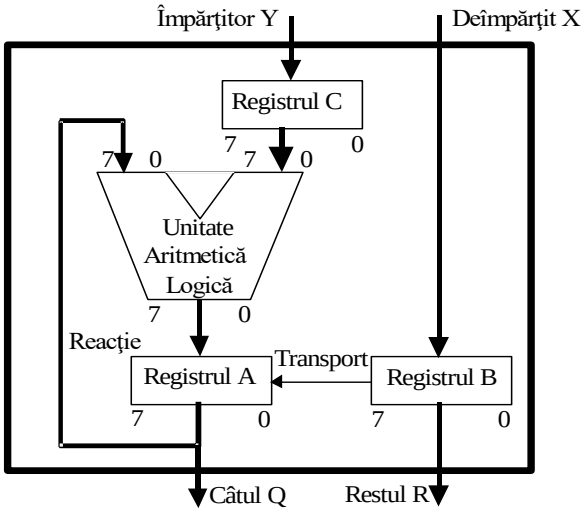

Figura 3.3 *Unitatea de execuţie a unui dispozitiv de împărţire binară (schemă de principiu)*

```
entity UNITATE_DE_EXECUŢIE is
 port (X, Y: in BIT_VECTOR (7 downto 0);
       Q,R: out BIT_VECTOR (7 downto 0));
-- X, Y, Q şi R sunt semnale externe
end entity UNITATE_DE_EXECUŢIE;
architecture UEA of UNITATE_DE_EXECUŢIE is
signal C: BIT_VECTOR (7 downto 0);
signal REACŢIE: BIT_VECTOR (7 downto 0);
signal TRANSPORT: BIT; --A,B: registre de deplasare pe 8 biţi
-- C, REACŢIE şi TRANSPORT sunt semnale interne
begin
-- Descrierea structurii interne a Unităţii de execuţie
...
end UEA;
```

Vizibilitatea fiecărui semnal este determinată de locul în care este declarat. Pentru determinarea vizibilităţii semnalelor se aplică următoarele reguli:

1. Un semnal declarat în cadrul unui pachet este vizibil în toate unităţile de proiectare care utilizează acel pachet;
2. Un semnal declarat ca port al unei entităţi este vizibil în toate arhitecturile asociate acelei entităţi;

3. Un semnal declarat în partea declarativă a unei arhitecturi este vizibil numai în interiorul acelei arhitecturi;
4. Un semnal declarat în cadrul unui bloc situat în interiorul unei arhitecturi este vizibil numai în interiorul acelui bloc.

Aceste reguli derivă direct din principiile proiectării ierarhice: dacă un obiect este declarat la un anumit nivel al ierarhiei, el va fi vizibil în interiorul acelei construcţii specifice şi în toate nivelurile inferioare ale ierarhiei. Figura 3.4 ilustrează aceste reguli într-un mod grafic:

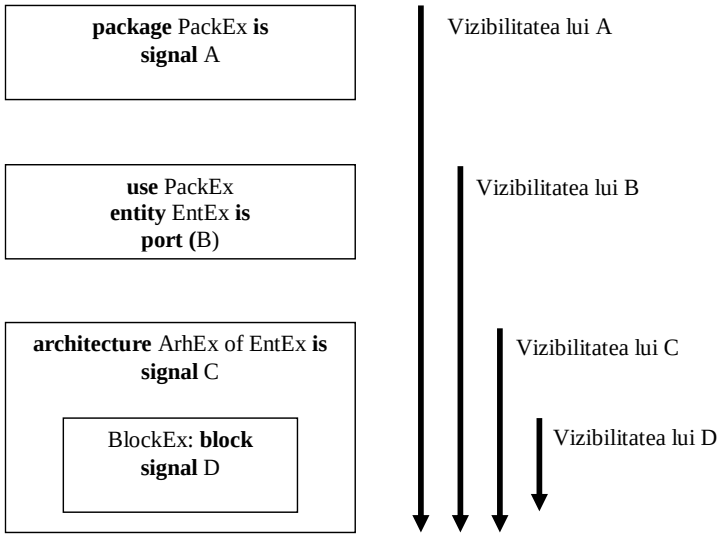

Figura 3.4 *Reguli de vizibilitate a semnalelor*

2.3 Asignarea de valori semnalelor

Instrucţiunile de asignare de valori semnalelor nu modifică valoarea actuală, ci valorile viitoare pe care aceste semnale le-ar putea lua – deci, modifică piloţii semnalelor (*driver*-ele).

Simbolul instrucţiunii de asignare este „<=” şi se citeşte „primeşte”. De exemplu, asignarea valorii semnalului B, semnalului A, se scrie astfel:

```
A <= B after 10 ns, '1' after 20 ns, '0' after 30 ns;
```

Această instrucţiune are următoarea semnificaţie: semnalul A va lua valoarea *actuală* a lui B peste 10 nanosecunde, apoi va lua valoarea ‘1’

peste 20 nanosecunde (deci cu 10 nanosecunde mai târziu) şi apoi valoarea ‘0’ peste încă 10 nanosecunde.

Fiecare pereche (valoare, întârziere), separată de cuvântul cheie **after**, este numită *element de formă de undă* (*waveform element*). Un şir de asemenea perechi constituie o *formă de undă* (*waveform*).

Valoarea trebuie să aibă un tip compatibil cu cel al semnalului căruia îi este asignată (în membrul stâng al instrucţiunii de asignare). Valoarea este fie o constantă, fie valoarea actuală a semnalului indicat sau, la modul general, rezultatul unei expresii.

Cuvântul cheie **after** este în mod obligatoriu urmat de către o expresie al cărei rezultat este de tipul TIME (tip definit în pachetul STANDARD). Rezultatul indică intervalul de timp la capătul căruia valoarea calculată (acum) va fi asignată semnalului. Sau, mai exact, *va fi probabil asignată*! În acest mod a fost calculată o posibilă valoare viitoare a semnalului, adică, cel mult, o valoare susceptibilă să devină valoare prezentă a semnalului. Acest viitor poate fi modificat prin acţiunea anumitor evenimente (datorate unor semnale interconectate sau unei noi situaţii apărute datorită avansului în timp).

Întârzierile specificate trebuie în mod obligatoriu să se prezinte în ordine crescătoare (în cazul de faţă 10, 20, 30). Această observaţie este esenţială pentru scrierea unei expresii (nu a unei constante) pentru calculul întârzierii. Nu există întârziere negativă în VHDL – definiţia tipului TIME din pachetul STANDARD nici nu o permite. Totuşi, este permisă utilizarea unei întârzieri nule.

Asignarea cu întârziere nulă se poate scrie astfel:

```
A <= B after 0 ns; -- Prima variantă
A <= B; -- A doua variantă; absenţa unei clauze after
-- semnifică o asignare a cărei întârziere este nulă
```

Atunci, care este semnificaţia unei întârzieri nule, de vreme ce nu se pune problema modificării valorii prezente a semnalului?

Întrucât limbajul VHDL este folosit pentru modelarea unor dispozitive hardware, noţiunea de asignare de valori cu întârziere nulă nu are sens: nu există dispozitive fizice care să nu prezinte timpi de propagare a semnalelor electrice prin ele, oricât de mici ar fi aceşti timpi. De fapt, asignarea precedentă semnifică pur şi simplu o cauzalitate: valoarea semnalului A este (potenţial) modificată din cauză că valoarea semnalului B se schimbă.

Iată în continuare modul în care simulatorul va traduce această cauzalitate pentru a efectua o simulare coerentă. Prezentăm noţiunea de *întârziere „delta*".

O întârziere „delta" este o întârziere care este nulă pentru simulare şi care nu reprezintă decât cauzalitatea.

Există aşadar *două dimensiuni ale timpului în VHDL*:

1. *Timpul „real"*, care se măsoară în paşi de simulare şi în unităţi de timp TIME. Acest timp este cel "văzut" de către proiectant.
2. *Timpul „delta"*, care este gestionat de către simulator. El îi permite acestuia să-şi aloce la fiecare pas al simulării un număr variabil de "felii" infinitezimale de timp şi să gestioneze astfel succesiunea asignărilor. Acesta reprezintă mijlocul de exprimare a „cauzalităţii" generate de către diferitele asignări.

Iată în continuare un exemplu de utilizare a întârzierilor „delta": fie un bistabil RS descris funcţional de următorul cod VHDL:

```
Q       <=      S nand NQ;
NQ      <=      R nand Q;
```

Să presupunem că bistabilul se află de un anumit timp în starea stabilă '1' (Q este '1', NQ este '0'), iar R şi S sunt '1' fiecare.

La un moment dat, notat cu T_0, intrarea R trece în starea '0'. În acest moment, se evaluează cea de-a doua instrucţiune de asignare, ceea ce face ca NQ să treacă în starea '1' la momentul T_0 + delta. Modificându-se NQ, prima instrucţiune de asignare va fi reevaluată, deci Q va lua valoarea '0' la momentul de timp $T_0 + (2 \times \text{delta})$. Deoarece această schimbare a valorii lui Q nu generează nici o modificare a lui NQ (care se află deja în starea '1'), evaluările iau sfârşit.

Aşa cum se arată în figura 3.5 a), au fost necesare mai multe intervale de timp „delta" pentru a descrie etapele succesive enumerate mai sus. Aceste intervale nu au o realitate funcţională din punctul de vedere al proiectantului (figura 3.5, b)).

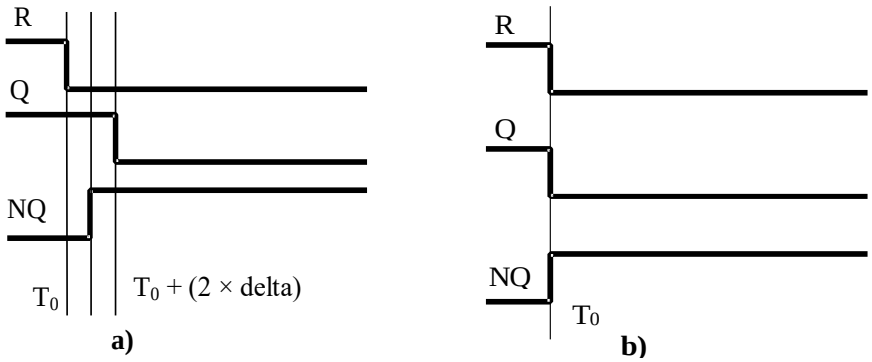

Figura 3.5 *Întârzierile „delta”: a) punctul de vedere al simulatorului; b) punctul de vedere al proiectantului*

Dacă proiectantul ar fi intenţionat să creeze un model mai apropiat de realitate, el ar fi trebuit să scrie (de exemplu):

```
Q     <=    S nand NQ after 5 ns;
NQ    <=    R nand Q after 5 ns;
```

Ceea ce ar fi determinat formele de undă din figura 3.6:

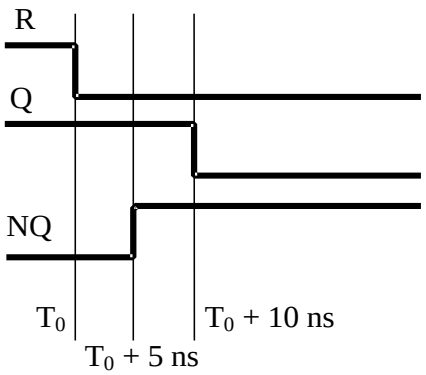

Figura 3.6 *Formele de undă aşteptate de către proiectant*

2.4 Modele de transmisie

Un sistem hardware poate prezenta un comportament de răspuns cu frecvenţă infinită – de exemplu, acesta este cazul unei linii obişnuite de transmisie de date. Orice impuls, indiferent de durata sa, va fi transmis.

Acest comportament este opus comportamentului sistemelor de comutaţie, la care apare un fenomen de inerţie: un impuls nu va fi transmis decât dacă durata sa va fi suficient de mare.

În VHDL, la asignarea fiecărui semnal, se poate specifica unul dintre următoarele două moduri de transmisie:

- *Modelul inerţial*. Acesta este modul de lucru implicit al oricărei instrucţiuni de atribuire de valori unui semnal, în care se filtrează impulsurile de durată inferioară timpului de transmisie.

```
SEMNAL1 <= REF after 10 ns;
```

Orice impuls prezent pe linia REF, a cărui durată este mai mică decât 10 nanosecunde, nu va fi transmis pe SEMNAL1;

- *Modelul transport*. Acesta reprezintă modelul cu răspuns de frecvenţă infinită. Este indicat de cuvântul cheie **transport** care urmează după simbolul operaţiunii de asignare. El permite transmiterea oricărui impuls, indiferent de durata acestuia.

```
SEMNAL2 <= transport REF after 10 ns;
```

Figura 3.7 ilustrează diferenţa dintre modelele de transmisie.

Începând cu VHDL'93 se poate face deosebirea dintre valoarea întârzierii (clauza **after**) şi lărgimea maximală a impulsurilor filtrate (clauza **reject**). În plus, cuvântul cheie **inertial** poate fi indicat în mod explicit în scopuri documentare, deşi rămâne modul implicit în VHDL.

Sintaxa generală pentru modelul de transmitere a semnalelor este deci următoarea:

```
[etichetă:] numele_semnalului <= [transport | [reject
lărgime_impuls] inertial] formă de undă;
```

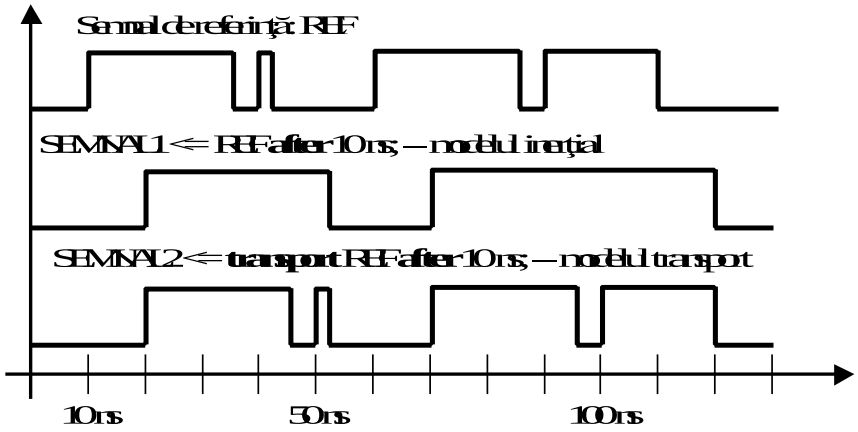

Figura 3.7 *Modelele de propagare inerţial şi transport*

Figura 3.8 ilustrează utilizarea clauzei **reject**. Dacă întârzierea introdusă rămâne cea indicată de clauza **after**, se vor filtra numai impulsurile de durată (sau „lărgime") inferioară timpului indicat de clauza **reject**. Se va genera o eroare dacă valoarea clauzei **reject** este mai mare decât cea a clauzei **after** (sau decât prima valoare a clauzei **after**, în cazul formelor de undă multiple).

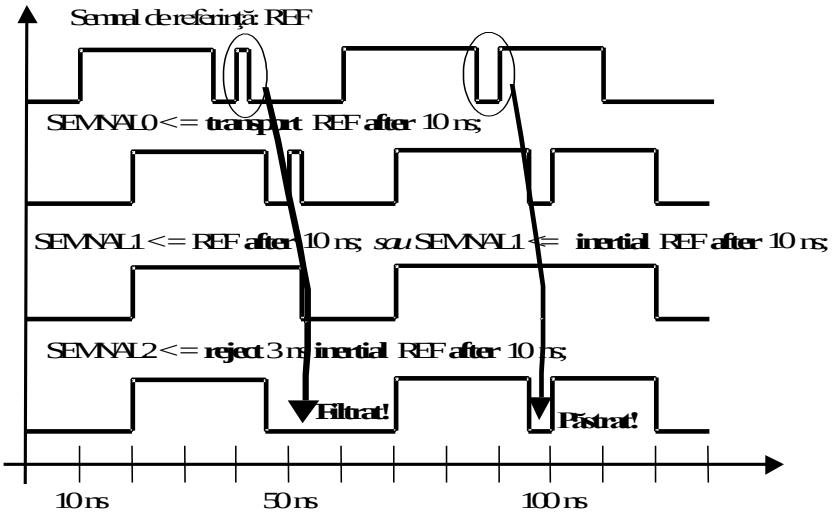

Figura 3.8 *Modelele de propagare cu lăţime de rejectare explicită*

Aceasta este o facilitate introdusă odată cu norma VHDL'93 şi nu o funcţionalitate suplimentară. Într-adevăr, instrucţiunea VHDL'93:

```
SEMNALX <= reject 6 ns inertial REF after 20 ns;
```

este echivalentă cu următoarele două linii (utilizând un semnal intermediar SEMNAL_INTER):

```
--Filtrarea inerţială a impulsurilor < 6 ns
SEMNAL_INTER <= REF after 6 ns;
-- Propagare pură (20 ns – 6 ns)
SEMNALX <= transport SEMNAL_INTER after 14 ns;
```

Sintaxa generală a instrucţiunii de asignare de semnal este foarte puternică: se poate utiliza un agregat atât în partea stângă a simbolului de asignare cât şi într-un element de formă de undă.

```
Nume_sau_agregat < = {transport} element_de_formă_de_undă 1,
element_de_formă_de_undă 2, …;
```

Un element de formă de undă se exprimă prin:

```
expresie {after expresie_temporală} -- Prima variantă
agregat {after expresie_temporală}  -- A doua variantă
```

Pentru a asigna valori unui agregat, adică pentru a-l putea folosi în membrul stâng al asignării, trebuie ca tipul agregatului să fie declarat şi ca identificarea acestui tip să se poată face fără ambiguitate din punctul de vedere al părţii drepte a instrucţiunii de asignare. Nu este permisă utilizarea unei expresii calificate (*tip_ obiect '(expresie)* sau *tip_obiect 'agregat*) în membrul stâng.

Presupunând că semnalele A şi B sunt de tipul BIT şi că se cunoaşte tipul BIT_VECTOR, am putea fi tentaţi să scriem:

```
(A, B) <= transport "10" after 10 ns;
```

Această instrucţiune nu va fi recunoscută în VHDL, membrul drept al instrucţiunii nefiind lipsit de ambiguitate. Într-adevăr, "10" poate la fel de

bine să fie de tipul BIT_VECTOR precum şi de tipul STRING, chiar dacă tipul STRING pare incoerent cu partea stângă. Scrierea corectă va fi:

```
(A, B) <= transport BIT_VECTOR'("10") after 10 ns;
-- Apare un apostrof! Expresia calificată (în membrul drept)
-- înlătură ambiguitatea.
```

Observaţie

La utilizarea unui agregat într-un element de formă de undă, aceleaşi restricţii interzic utilizarea cuvântului cheie **others**; tipul agregatului din stânga trebuie să poată fi identificat fără ambiguitate.

2.5 Programe cu execuţie secvenţială

Rularea secvenţială a unui program este uşor de înţeles, deoarece se ştie exact care este instrucţiunea executată de către program. Instrucţiunile se succed într-o ordine precisă şi nu trebuie făcut nici un calcul pentru determinarea instrucţiunii executate. Modul de rulare secvenţial este caracteristic majorităţii limbajelor de programare clasice.

În VHDL însă, în general, instrucţiunile se execută concurent, cu excepţia anumitor părţi care se scriu şi se execută în manieră secvenţială (de exemplu, corpul unui proces). Semnalele se vor comporta în moduri diferite în funcţie de tipul lor de propagare: inerţial sau transport.

Fie două tranzacţii, numite „prima” şi „a doua” – după ordinea lor de apariţie într-un proces.

	TRANSPORT	**INERŢIAL**
A doua tranzacţie are loc **înaintea** primei tranzacţii.	Scrie peste prima tranzacţie	Scrie peste prima tranzacţie
A doua tranzacţie are loc **după** prima tranzacţie.	Adaugă a doua tranzacţie la semnal	Scrie peste prima tranzacţie dacă valorile sunt diferite; dacă nu, ambele sunt păstrate

Următoarele patru figuri vor ilustra efectele codificării VHDL.

1a) Inerţial – tranzacţia 2 are loc înaintea tranzacţiei 1 (figura 3.9).

```
begin
      process
      begin
            IEŞIRE <= '1' after 10 ns; -- prima tranzacţie
            IEŞIRE <= '0' after 5 ns; -- a doua tranzacţie
            wait;
      end process;
end;
```

1b) Transport - tranzacţia 2 are loc înaintea tranzacţiei 1 (figura 3.9).

```
begin
 process
 begin
  IEŞIRE <= transport '1' after 10 ns; -- prima tranzacţie
  IEŞIRE <= transport '0' after 5 ns; -- a doua tranzacţie
  wait;
 end process;
end;
```

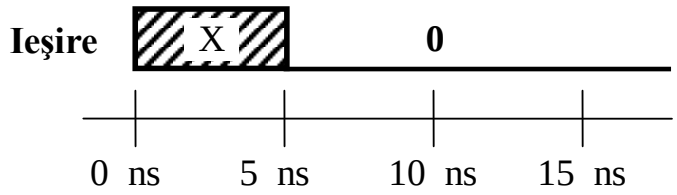

Figura 3.9

2) Inerţial – tranzacţia 2 are loc după tranzacţia 1, valori similare (figura 3.10).

```
begin
      process
      begin
            IEŞIRE <= '0' after 10 ns; -- prima tranzacţie
            IEŞIRE <= '0' after 15 ns; -- a doua tranzacţie
            wait;
      end process;
end;
```

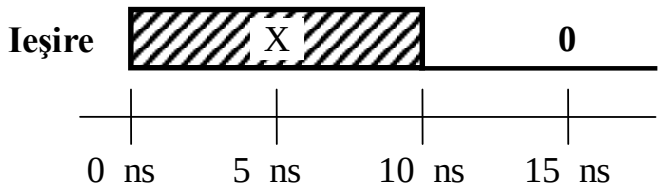

Figura 3.10

3) Inerţial – tranzacţia 2 are loc după tranzacţia 1, valori diferite (figura 3.11).

```
begin
      process
      begin
            IEŞIRE <= '1' after 10 ns; -- prima tranzacţie
            IEŞIRE <= '0' after 15 ns; -- a doua tranzacţie
            wait;
      end process;
end;
```

Starea '1' care ar fi trebuit să apară între t = 10 ns şi t = 15 ns a fost filtrată (suprimată).

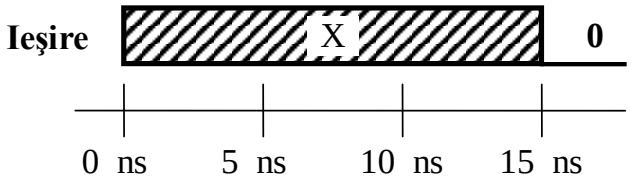

Figura 3.11

4) Transport – tranzacţia 2 are loc după tranzacţia 1 (figura 3.12)

```
begin
 process
 begin
  IEŞIRE <= transport '1' after 10 ns; -- prima tranzacţie
  IEŞIRE <= transport '0' after 15 ns; -- a doua tranzacţie
  wait;
 end process;
end;
```

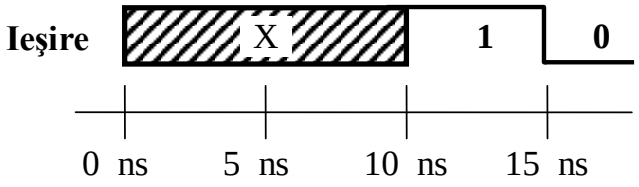

Figura 3.12

2.6 Parametri generici

Utilizarea parametrilor generici reprezintă mijlocul de a transmite o informaţie unui bloc, această informaţie fiind *statică* pentru blocul respectiv.

Un bloc generic este văzut din exterior ca un bloc parametrizat, prin intermediul *parametrilor generici*. Din interiorul blocului, aceşti parametri sunt văzuţi drept constante şi pot fi manipulaţi ca atare.

De exemplu, se poate descrie un bloc cu un comportament de registru pe N biţi, valoarea lui N fiind stabilită de un parametru generic.

Blocurile generice sunt foarte utilizate în VHDL. Bibliotecile de modele sunt de foarte multe ori generice; în general se urmăreşte ca aceste biblioteci să conţină componente generale.

Obiectele care pot fi generice în VHDL sunt următoarele:

1. *O entitate* (o pereche entitate / arhitectură). O asemenea unitate de proiectare poate fi compilată (dar nu şi simulată ca atare!) şi poate fi regăsită în bibliotecă. De reţinut că o entitate poate fi generică, dar o arhitectură – nu!;
2. *Orice bloc intern* (definit cu ajutorul instrucţiunii **block**). Totuşi, acest bloc nu poate fi instanţiat nici din exteriorul arhitecturii în care se găseşte, nici din interior. Blocul intern generic nu poate fi instanţiat decât în momentul declarării sale (fie cu parametri de intrare, fie cu variabile). Această restricţie face ca utilizarea blocurilor generice să fie dificilă în cadrul modelării în VHDL. Raţiunea sa de a fi o constituie echivalenţa a două blocuri interne imbricate cu o entitate, aceasta din urmă putând fi generică.

Parametrii generici pot fi folosiţi oriunde în cod, acolo unde este necesară o valoare statică. De fapt, se recomandă insistent să se utilizeze parametrii generici şi constantele în locul codificării fixe, deoarece uşurează modificarea proiectului. În general, parametrii care se declară generici sunt:

- *Dimensiunea* obiectelor complexe, cum ar fi vectorii sau magistralele. În cazul utilizării parametrilor generici pentru definirea lăţimii de bandă a unei magistrale, se poate modifica

clauza **generic** de la începutul specificaţiei. Parametrii generici se pot folosi ca limite de interval pentru iteratorii buclelor **for** şi pentru alte aplicaţii similare;

- *Parametrii de temporizare*: întârzieri, timpi de prepoziţionare (*set-up*), timpi de menţinere (*hold*), timpi de comutare şi orice alte caracteristici temporale ale dispozitivelor electronice.

Declaraţia unui bloc generic constă, din punct de vedere sintactic, în scrierea unei clauze **generic** după antetul blocului:

```
generic (parametru_1 {, alt_parametru} : tipul_ parametrului
{:= valoarea_implicită};
        (parametru_2 {, alt_parametru} : tipul_ parametrului
{:= valoarea_implicită};
         ...
        (parametru_n {, alt_parametru} : tipul_ parametrului
{:= valoarea_implicită};
```

Ca tip al parametrului se poate indica un tip de date efectiv, dar şi, la modul general, indicaţii de sub-tipizare. Valoarea implicită este valoarea care va fi utilizată în momentul instanţierii blocului, dacă parametrul corespunzător este omis.

Această sintaxă se aplică atât entităţilor (numite *blocuri externe*) cât şi instrucţiunilor bloc (numite *blocuri interne*). Ea este adeseori urmată de o clauză (opţională) ce va descrie porturile acestui bloc (cuvântul cheie **port**).

Iată un exemplu în care parametrul generic este direct utilizat în descrierea porturilor. Se urmăreşte descrierea unei vederi externe a unei porţi logice ŞI cu N intrări:

```
entity POARTA_ŞI is
       generic (NUMĂR_DE_INTRĂRI: NATURAL := 2);
       port (INTRĂRI: in BIT_VECTOR (1 to NUMĂR_DE_INTRĂRI);
             IEŞIRE: out BIT);
end POARTA_ŞI;
```

Aceeaşi funcţionalitate, aplicată unui bloc intern, se scrie astfel:

```
POARTA_ŞI: block
 generic (NUMĂR_INTRĂRI: NATURAL := 2);
 port (INTRĂRI: in BIT_VECTOR (1 to NUMĂR_INTRĂRI);
```

```
       IEŞIRE: out BIT);
 generic map (NUMĂR_INTRĂRI => 8); -- Instanţierea unei porţi
                                   -- ŞI cu 8 intrări
begin  --Zona de instrucţiuni din cadrul blocului
       process (INTRĂRI)
        variable V: BIT:='1';
        begin
        for I in 1 to NUMĂR_INTRĂRI loop
         V:= V and INTRĂRI(I);
        end loop;
        IEŞIRE <= V;
       end process;
end POARTA_ŞI;
```

Să presupunem de exemplu că dorim să descriem o componentă generală capabilă să recunoască un profil oarecare de N biţi transmişi în serie. Figura 3.13 prezintă schema de principiu a dispozitivului.

Secvenţa de recunoscut se prezintă la intrarea INTRARE, fiind sincronizată de semnalul de TACT. Detectarea secvenţei provoacă trecerea ieşirii DETECTAT în starea '1' logic, aceasta fiind '0' în toate celelalte cazuri.

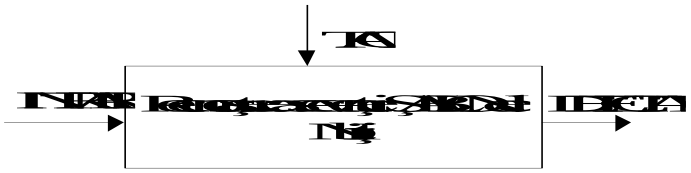

Figura 3.13 *Dispozitiv de detectare a unei secvenţe binare*

Datorită utilizării parametrilor generici, entitatea se poate scrie:

```
entity RECUNOAŞTERE is
       generic (SECVENŢA: BIT_VECTOR);
       port (INTRARE, TACT: in BIT;
             DETECTAT: out BIT);
end RECUNOAŞTERE;
```

Una dintre arhitecturile posibile va fi:

```
architecture SIMPLĂ of RECUNOAŞTERE is
signal MEMORIE: BIT_VECTOR (SECVENŢA'RANGE);
begin
-- Conversia unui BOOLEAN în tipul BIT. Atributele VAL şi POS
```

```
-- sunt studiate în lucrarea 5.
DETECTAT <= BIT'VAL(BOOLEAN'POS(MEMORIE = SECVENŢA));
MEMORARE: process
begin
if TACT = '1' then
MEMORIE<=MEMORIE(MEMORIE'LEFT+1 to MEMORIE'RIGHT) & INTRARE;
end if;
wait on TACT;
end process MEMORARE;
end SIMPLĂ;
```

Observaţii

1. Testul TACT = '1' e utilizabil deoarece TACT este de tipul BIT. Dacă, de exemplu, TACT ar putea lua valorile 'X', '0', '1' sau 'Z', acest test ar fi adevărat (în mod greşit) în decursul trecerilor de la 'X' la '1' sau de la 'Z' la '1'. Ar trebui deci regândită expresia utilizând atribute adecvate.
2. Descrierea de mai sus se pretează foarte bine la o transformare în arhitectură recursivă.

2.7 Constante

Constantele joacă acelaşi rol ca şi parametrii generici: ele constituie un suport al informaţiei statice care poate fi utilizată în interiorul unui model. Cu toate acestea, spre deosebire de parametrii generici (care sunt declaraţi în *entităţi*), constantele sunt declarate în cadrul *arhitecturilor*.

Constanta este un obiect de bază în VHDL. O constantă este iniţializată la o valoare care nu mai poate fi modificată ulterior. Valoarea de iniţializare trebuie să fie de acelaşi tip ca şi cel indicat în momentul declarării constantei. Ea va fi calculată în etapa de elaborare şi poate fi o expresie complexă, incluzând eventual apeluri de funcţii (putând chiar să preia o valoare de la tastatură, de exemplu). Sintaxa sa este următoarea:

```
constant nume_constantă: tip_sau_sub-tip {:= valoare_constantă};
```

Iată câteva exemple de declaraţii de constante:

```
constant PI: REAL := 3.1416;
constant ÎNTÂRZIERE_MAXIMĂ: TIME := 25 ns;
constant MARJĂ_MIN: INTEGER := 40*N; -- N e definit anterior
```

În ultimul exemplu, N poate desemna un parametru de intrare al procedurii în care se află această declaraţie şi poate deci conferi un caracter „dinamic" valorii constantei MARJĂ_MIN.

O constantă nu poate avea nici tipul **acces** (nici un alt tip mai complex care are un tip **acces** printre elementele sale), nici tipul fişier (**file**).

Ţinând cont de sintaxa precedentă, valoarea constantei poate să nu fie precizată în momentul declaraţiei sale. Utilitatea acestei opţiuni poate să nu pară evidentă. În practică, o declaraţie de constantă fără valoarea sa se numeşte *declaraţie de constantă cu iniţializare ulterioară*.

Acest tip de declaraţie de constantă nu are sens decât dacă se găseşte în partea de specificare a unui pachet. Scopul său este simplu: din exteriorul pachetului se poate folosi această constantă, dar fără a avea cunoştinţă de valoarea sa care este definită printr-o declaraţie completă de constantă în interiorul corpului pachetului. Astfel, un proiectant care va utiliza această constantă nu va putea fi tentat să o înlocuiască prin valoarea sa: el nu o cunoaşte. Acesta este un exemplu de mascare a informaţiei în VHDL.

Exemplul de mai jos ilustrează utilizarea constantelor:

```
entity MUX_2_LA_1 is
       port (I0, I1, SEL: in BIT;
             Y: out BIT):
end entity MUX_2_LA_1;
architecture ARHITECTURA of MUX_2_LA_1 is
constant TP: TIME := 10 ns;
begin
       Y <= (I0 and SEL) or (I1 and not (SEL)) after TP;
end ARHITECTURA;
```

Ne putem întreba de ce conţine limbajul VHDL două construcţii atât de similare cum ar fi parametrii generici şi constantele. Este, oare, într-adevăr utilă existenţa ambelor?

Principala diferenţă dintre *parametri generici* şi *constante* constă în aceea că parametrii generici pot fi utilizaţi şi în mod *dinamic*, în vreme ce constantele sunt pur *statice*. Asta înseamnă că valoarea unui parametru generic poate fi modificată fără a se opera modificări în cod, deci fără a se recompila entitatea. Dimpotrivă, constantele nu pot fi modificate fără a se opera modificări în cadrul codului. Acest aspect este important mai ales în cazul în care specificaţia va fi folosită ca o *componentă* pentru o altă specificaţie, de nivel superior. De fiecare dată când această componentă este utilizată, i se pot atribui noi valori, dacă ele sunt specificate prin parametri generici.

Aceeaşi entitate poate fi partajată de mai multe arhitecturi şi toţi parametrii generici se aplică tuturor arhitecturile entităţii. De vreme ce acelaşi nume de parametru generic poate fi folosit în mai multe arhitecturi, orice schimbare a valorii sale va afecta toate instanţele sale din toate arhitecturile. În schimb, dacă se folosesc constante, acestea vor localiza schimbările numai în arhitectura selectată.

3. Desfăşurarea lucrării

3.1 Se vor crea entităţile Memorie_ROM şi Memorie_RAM (conform figurii 3.2).

3.2 Se vor descrie prin arhitecturi structurile interne pentru memoriile ROM şi RAM, utilizând semnale. Harta memoriilor va fi cea necesară implementării următoarelor funcţii: $f_1 = \Sigma$ (0, 2, 8, 10); $f_2 = \Sigma$ (0, 1, 2, 5, 6, 8, 10); $f_3 = \Sigma$ (1, 2, 4, 5, 7, 11); $f_4 = \Sigma$ (0, 2, 8, 9, 13, 15);

3.3 Se vor testa modelele de transmitere ale semnalelor pentru semnalele definite în arhitecturile de la punctul 3.2.

3.4 Se vor implementa şi testa exemplele POARTA_ŞI şi MUX_2_LA_1 descrise în lucrare pentru înţelegerea utilizării parametrilor generici.