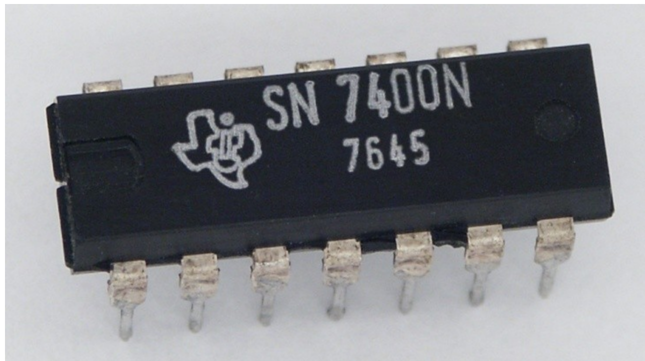
INTRODUCERE IN LIMBAJUL DE DESCRIERE HARDWARE VERILOG

1. Limbaje de descriere hardware – privire de ansamblu

Clasificarea circuitelor integrate digitale in functie de complexitate:

- SSI (*Small Scale Integration*) - primele circuite digitale, aprox. 1960
  - contineau **zeci** de tranzistori/porti logice



a)

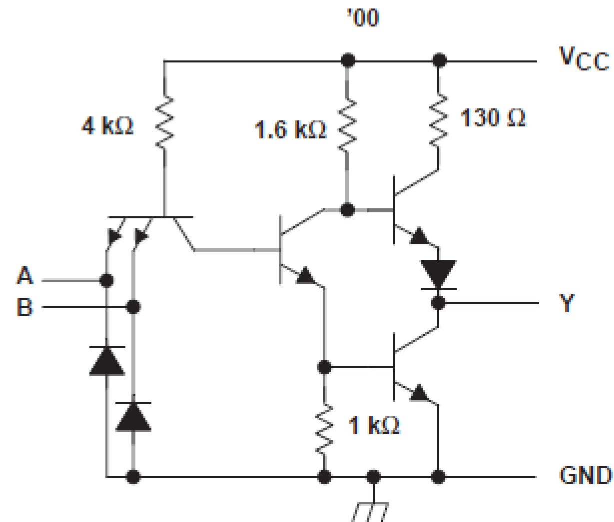


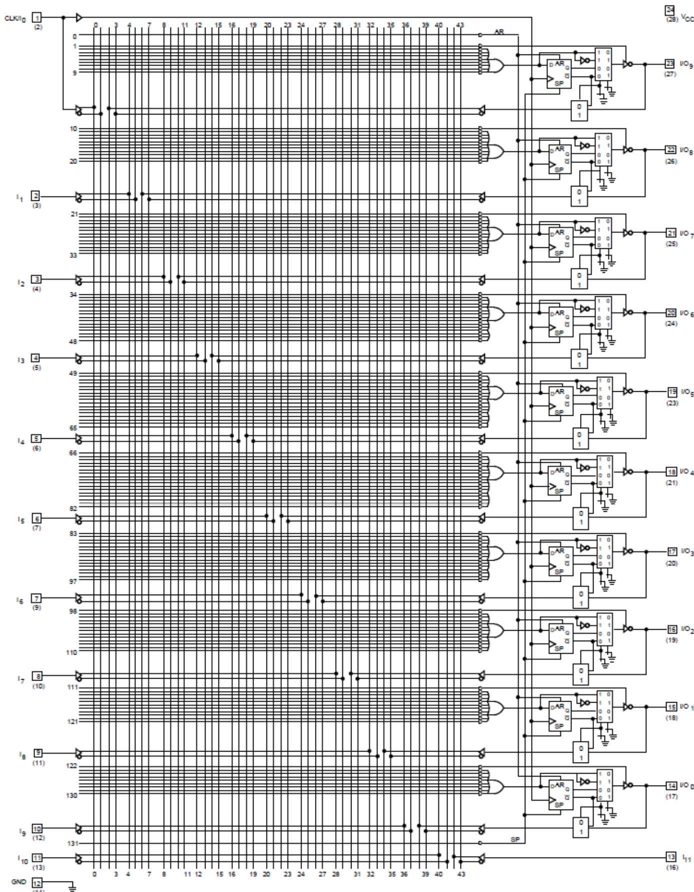
Fig. 1.1 Circuitul digital cu patru porti NAND de tip SN7400

a) Capsula circuitului

b) Schema unei porti NAND

- MSI (*Medium Scale Integration*) - **sute** de porti logice pe un chip
  - sfarsitul anilor 1960

  Exemplu: PLA (*Programmable Logic Array*) – Arie Logica Programabila



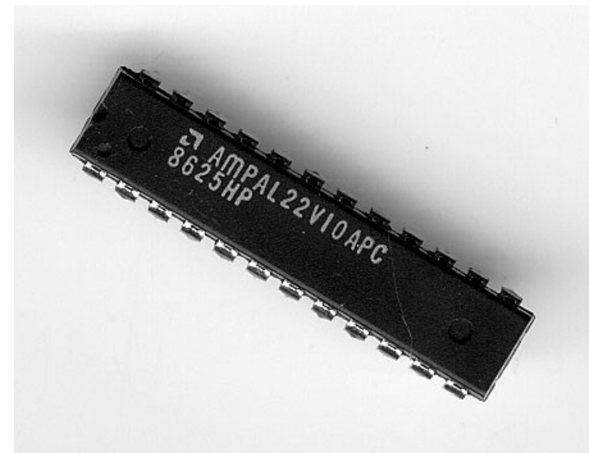
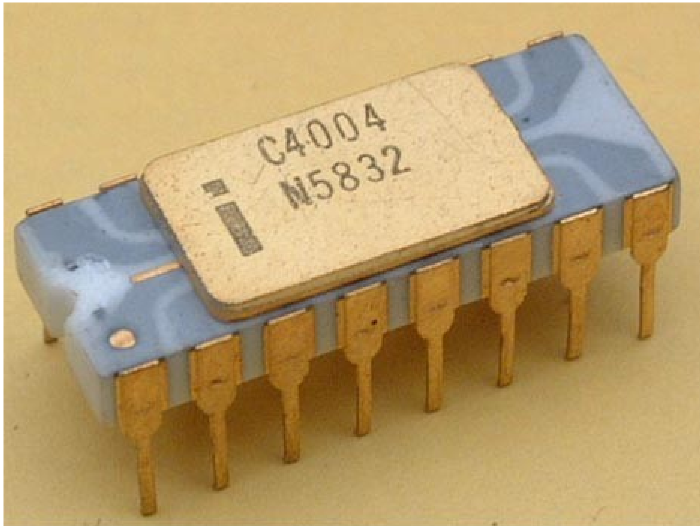


Fig. 1.2 Circuitul programabil produs de firma AMD (1983)

- LSI (*Large Scale Integration*) - **mii** de porţi logice pe un singur chip, ~1970
    - proiectarea manuală a schemelor logice devine foarte greoaie, se doreşte automatizarea
    - apar uneltele CAD (*Computer Aided Design*)
    - se pot verifica prin simulare blocuri de ~100 porţi logice, circuitele finale încă se testează pe “bancuri de probă” (*breadboard*)



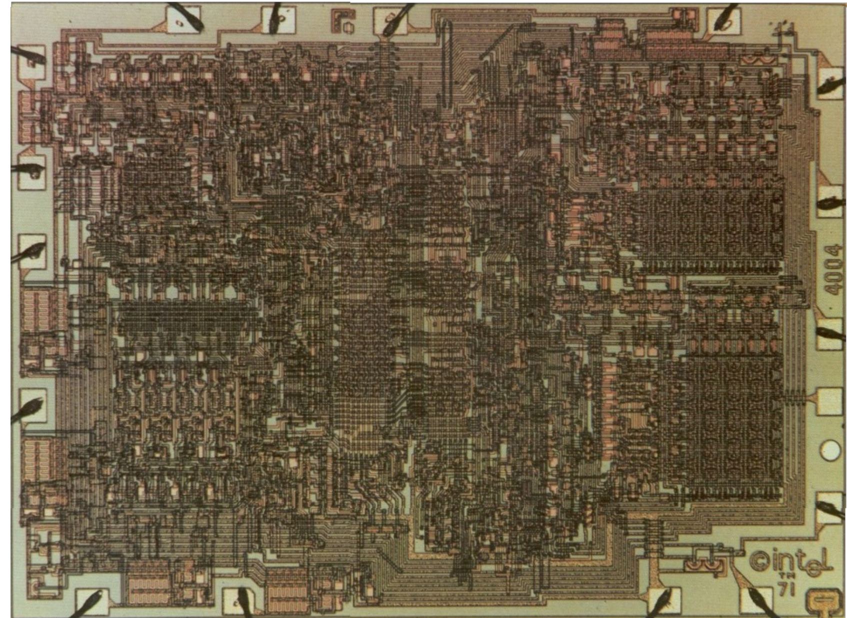


Fig. 1.3 Primul microprocesor Intel 4004 – 2300 tranzistori (1971)

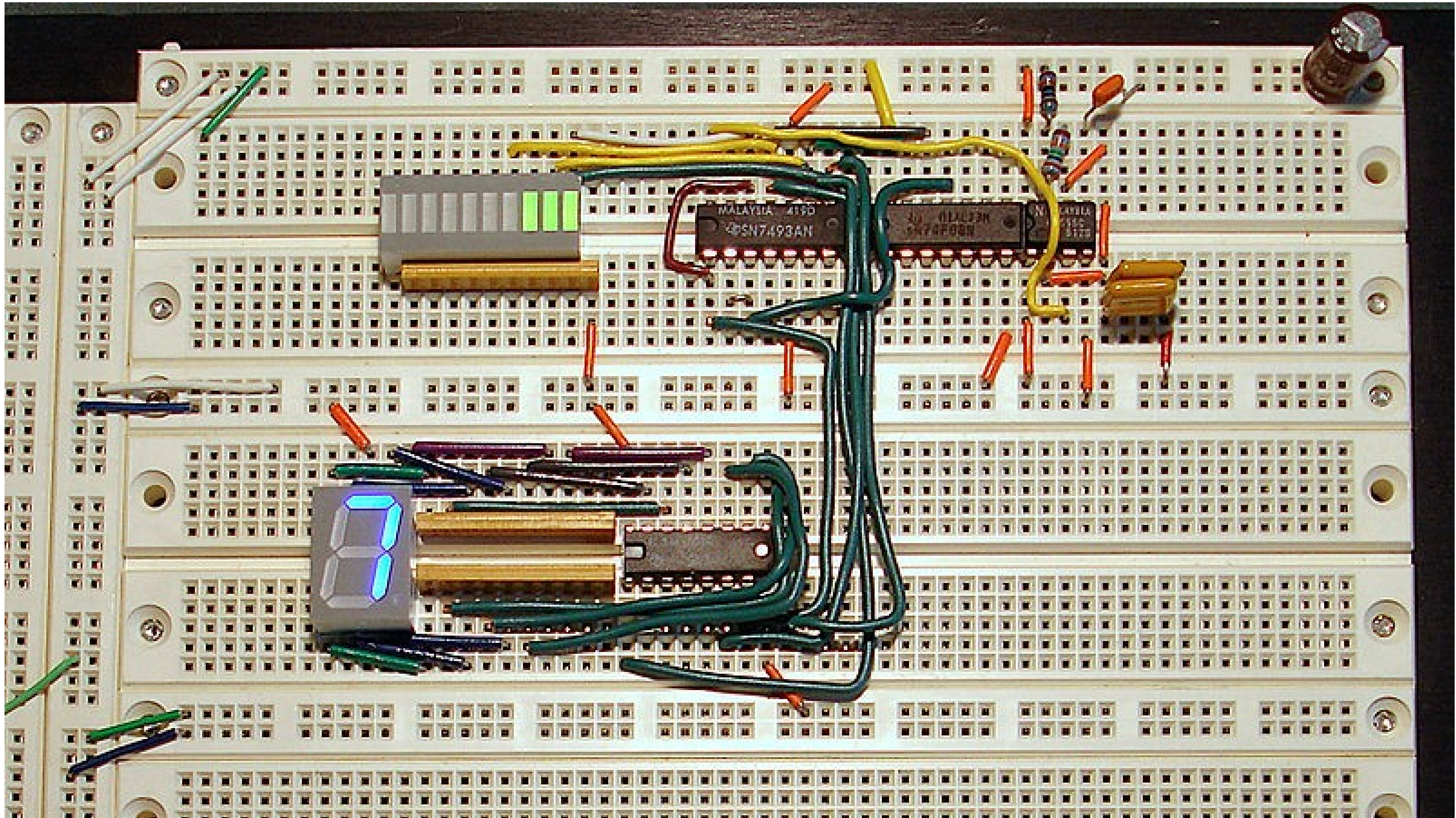


Fig. 1.4 Exemplu de banc de probă *(breadboard)*

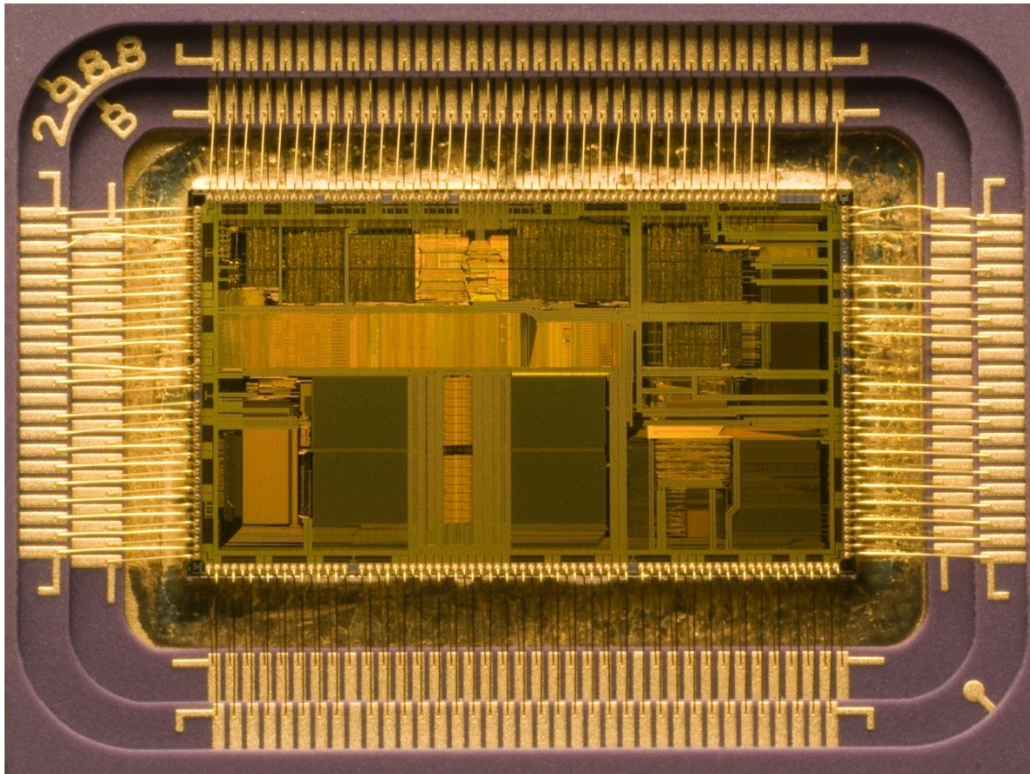
- VLSI (*Very Large Scale Integration*) - numarul de tranzistori integrati pe un chip a depasit **100.000**

folosirea “bancului de proba” devine aproape imposibila

uneltele CAD devin instrumente vitale pentru proiectare (instrumente automate pentru amplasarea si interconectarea automată a circuitelor logice)

se pot proiecta circuite digitale grafic, cu ajutorul porţilor logice; blocurile rezultate sunt folosite mai departe ca unitati functionale in scheme din ce in ce mai complexe

Exemplu: microprocesorul pe 32 biti Intel 80486 DX2 - 1.180.000 tranzistori



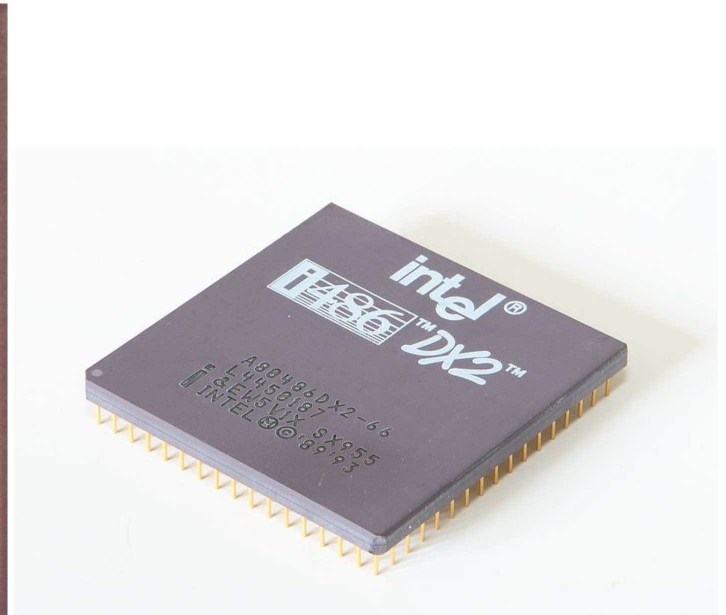


Fig. 1.5 Microprocesorul pe 32 biti Intel 80486 DX2 (1989)

Complexitatea tot mai mare a circuitelor a condus la inventarea unor limbaje pentru a descrie circuitele digitale, particularitatea lor fiind capacitatea de a abstractiza paralelismul intalnit in circuitele fizice => **Limbajul de descriere hardware** <=>*Hardware Description Language* (HDL)

Limbajele HDL cele mai folosite sunt Verilog si VHDL, ambele au inceput sa fie dezvoltate incepand cu anii 1980.
Verilog <= *Gateway Design Automation* (*Cadence Systems*) 1983
VHDL <= DARPA (Defense Advanced Research Project Agency)

Folosite la inceput numai pentru verificarea din punct de vedere logic a circuitelor digitale, spre sfarsitul anilor 1980, aparitia tehnologiei *sintezei logice* a produs o schimbare radicala in tehnica de proiectare. Circuitele digitale puteau fi descrise prin intermediul *transferului intre registre*, detaliile de implementare fizica fiind lasate pe seama sistemului automat CAD.

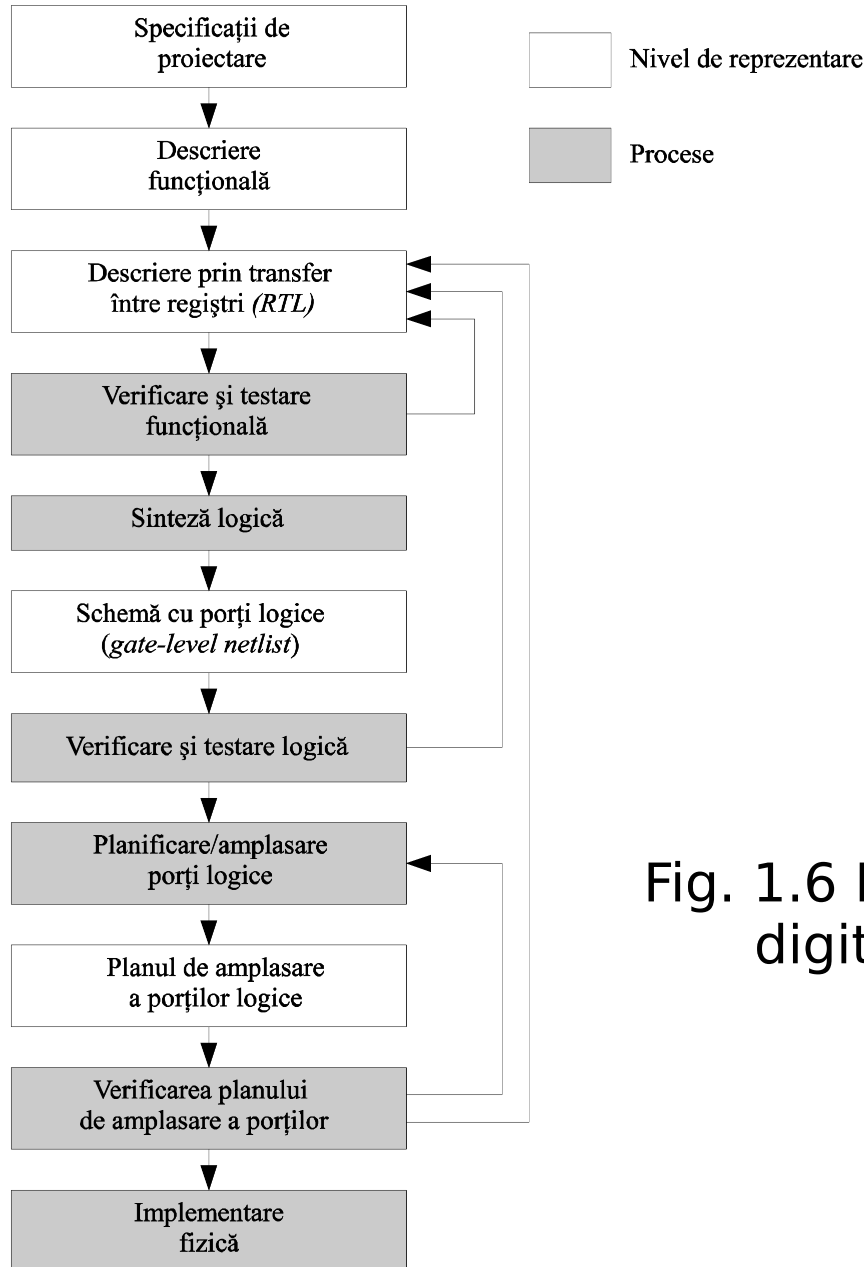


Fig. 1.6 Etapele realizării unui circuit digital cu ajutorul limbajelor HDL

Etapele parcurse de proiectanţi pentru realizarea unui circuit digital cu ajutorul limbajelor HDL:

- Enuntarea specificatiilor circuitului digital insotita de o descriere functionala care precizeaza performantele, compatibilitatea cu standardele existente etc.
- Conversia manuala a descrierii realizate intr-o alta descriere prin *transfer intre registri*;
- Proiectarea circuitului devine din aceasta etapa automatizata, generarea schemei de conexiuni intre portile logice si amplasarea lor facandu-se cu ajutorul uneltelor CAD. Rezultatul final al proiectarii, schema de amplasare finala, poate fi transpusa direct pe cipuri de siliciu.

! De retinut ca desi instrumentele CAD simplifica mult proiectarea unui circuit digital, folosirea lor in orice conditii poate determina un rezultat total ineficient, ramanand proiectantului sarcina de a “ghida” procesul automat pentru obtinerea maximului de performanta.

Avantajele limbajelor de descriere hardware in comparatie cu metodele clasice de proiectare:

- Circuitul digital fiind proiectat la un inalt nivel de abstractizare se obtine independenta de tehnologia prin care este realizat fizic circuitul, noile tehnologii putand fi asimilate cu ajutorul noilor unelte de sinteza disponibile;

- Posibilitatea verificarii din punct de vedere logic si testarii performantelor circuitului digital inca din faza de cod HDL elimina practic posibilitatea aparitiei unor erori in fazele ulterioare de dezvoltare ale cipului;

- Pentru proiectanti/programatori descrierea textuala a unui circuit digital complex poate fi mai concisa si mai simplu de depanat decat lucrul cu o schema la nivel de porti logice care poate deveni extrem de complicată

Particularitati ale limbajului Verilog

1. Posibilitatea scrierii in acelasi model a unui cod de nivel inalt si a unui cod de nivel scazut de abstractizare
2. Asemanarea cu limbajul de programare procedurala C
3. Existenta multor instrumente software atat comerciale cat si gratuite

4. Cei mai importanti fabricanti de circuite logice digitale ofera biblioteci Verilog pentru sinteza logica

5. Interfata de programare PLI (*Programming Language Interface*) este un instrument util pentru interfatarea codului scris in limbajul C cu structurile de date interne Verilog

2. Modelarea ierarhică

Exista doua abordari in proiectarea circuitelor digitale:

→ Abordarea de la nivel inalt la nivel scazut (*top-down*)

Proiectarea incepe cu stabilirea unui bloc functional cat mai general, care va necesita divizarea in mai multe sub-blocuri functionale pana se ajunge la limita de divizare (ex. poarta logica)

→ Abordarea de la nivel scazut la nivel inalt *(bottom-up*)

In acest caz proiectarea incepe cu identificarea blocurilor atomice (care nu mai pot fi divizate, de exemplu portile logice) disponibile, cu ajutorul lor urmand sa se construiasca blocuri din ce in ce mai complexe pana se atinge functionalitatea dorita a circuitului digital

2.1 Modelarea unui numarator asincron pe 4 biti, declansat pe front negativ

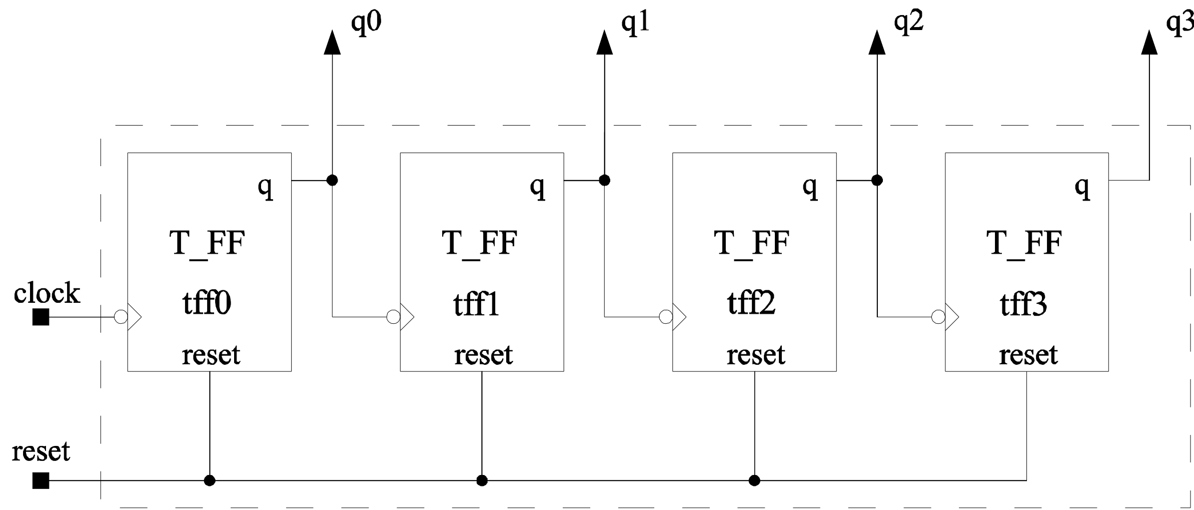
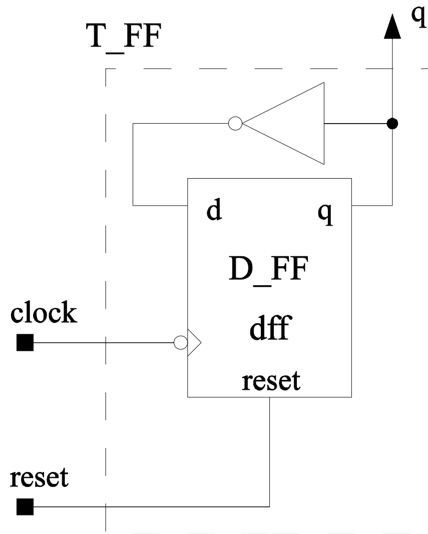


Fig. 2.1 Schema bloc a unui numarator asincron pe 4 biti

Numaratorul prezentat in figura 2.1 este format din patru bistabili *(Flip-Flop)* de tip T conectati in cascada, la care intrarile de ceas sunt conectate la iesirile bistabililor anteriori. Bistabilul de tip T este realizat la randul lui dintr-un bistabil de tip D si o poarta de tip NOT, conectata ca in figura 2.2. In acest caz este folosita abordarea *top-down*.



Bistabil de tip T			Bistabil de tip D		
reset	q	q+1	reset	d	q+1
1	0	0	1	0	0
1	1	0	1	1	0
0	0	1	0	0	0
0	1	0	0	1	1

a) Schema bistabilului T b) Tabelul de adevar pentru bistabilele T si D

Fig. 2.2 Bistabilul de tip T construit cu ajutorul unui bistabil de tip D

In cazul unei abordari *bottom-up* se vor construi bistabili cu ajutorul portilor AND, OR , NOR, NAND sau chiar al tranzistorilor.

2.3 Modulul - Unitatea functionala a programelor in limbajul Verilog

Modulul reprezinta unitatea atomica din care se compune orice program Verilog. Acesta poate fi de sine statator sau avand in componenta alte module. Modulul pune la dispoziţia utilizatorului o interfata numita port de intrare/ieşire, astfel mascand codul aflat in interiorul sau.

Modulul Verilog se codifica folosind cuvantul cheie **module s**i perechea **endmodule**. Fiecare modul poseda un nume de indentificare si o lista de porturi intrare/iesire.

```
module <nume_modul> (<lista_de_porturi>);
. . .
<cod Verilog>
endmodule
```

Exemplu pentru bistabilul T:

```
module T_FF (q, clock, reset);
. . .
. . .
endmodule
```

Modulul reprezinta un sablon *(template)* cu ajutorul caruia se pot crea obiecte separate, avand nume propriu, variabile, parametri si interfata I/O. Procesul de creare de obiecte din sabloane se numeste **instantiere**, iar obiectele **instante**.

Exemplu de instantiere:

```
module counter4bit(q,clk,reset);
    output [3:0] q;
    input clk, reset;

    T_FF tff0(q[0],clk,reset);
    T_FF tff1(q[1],q[0],reset);
    T_FF tff2(q[2],q[1],reset);
    T_FF tff3(q[3],q[2],reset);
endmodule
```

Programul de mai sus presupune ca un modul numit T_FF avand porturile specificate a fost definit anterior.

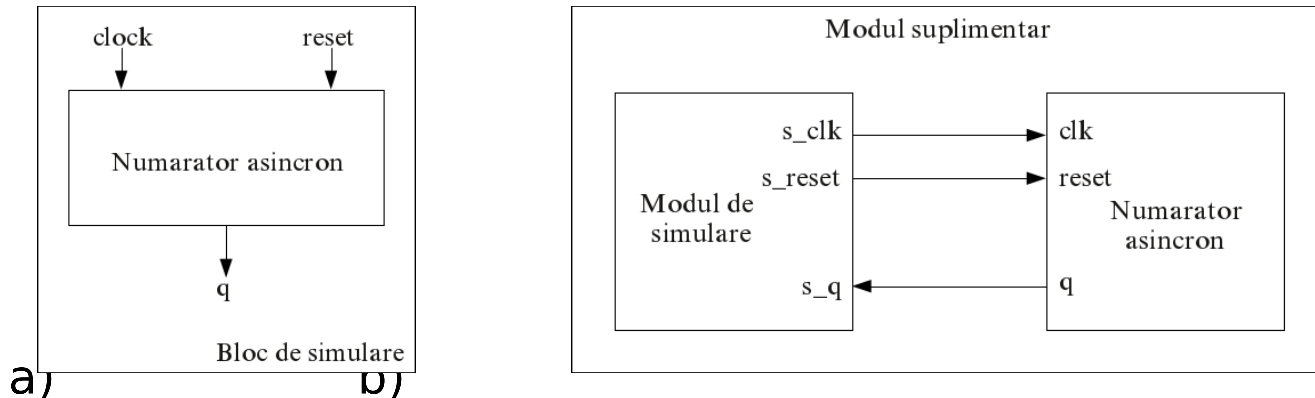
In limbajul Verilog nu se accepta imbricarea modulelor, nu se poate definì un modul intr-un alt modul.

Exemplu de cod GRESIT!

```
module num4bit(q, clk, reset);
    output [3:0] q;
    input clk, reset;

    module T_FF(q, clk,reset);//MODUL DEFINIT GRESIT
        . . .
    endmodule
endmodule
```

3. Simularea modulelor Verilog

Orice circuit digital proiectat cu ajutorul unui limbaj HDL trebuie verificat/simulat, in acest scop se utilizeaza un bloc de simulare. Blocul de simulare sau “bancul de proba” (test bench) este la randul lui proiectat cu ajutorul limbajului HDL.



a) b)

Fig. 3.1 Simularea unui modul Verilog: a) prin instantiere, b) prin conectare

Numarator asincron pe 4 biti: modul de implementare si modul de simulare – abordare de tip *top-down*

```
//modulul final pentru numaratorul pe 4 biti
module num4bit(q,clk,reset);
    output [3:0] q;
    input clk,reset;

    T_FF tff0(q[0],clk,reset);
    T_FF tff1(q[1],q[0],reset);
    T_FF tff2(q[2],q[1],reset);
    T_FF tff3(q[3],q[2],reset);
endmodule


//modulul pentru bistabilul T
module T_FF(q,clk,reset);
    output q;
    input clk,reset;
    D_FF dff(q,d,clk,reset);
    not n1(d,q);
endmodule
```

```
//modulul pentru bistabilul D
module D_FF(q,d,clk,reset);
   input  d,clk,reset;
   output q;
   reg q;
always @(posedge reset or negedge clk)
   if(reset)
      q=1'b0;
   else
      q=d;
endmodule
```

Modulul de simulare – de tip instantiere

```
//modul de simulare
module stimulus;
   reg clk,reset;
   wire[3:0] q;
   num4bit num(q,clk,reset);  //instantiere modul numarator
   initial
      clk=1'b0;
```

```
always
   #5 clk=~clk;

initial
   begin
          reset=1'b1;
   #15    reset=1'b0;
   #180   reset=1'b1;
   #10    reset=1'b0;
   #20    $finish;
   end
initial
   $monitor($time,” q=%d”,q);
endmodule
```

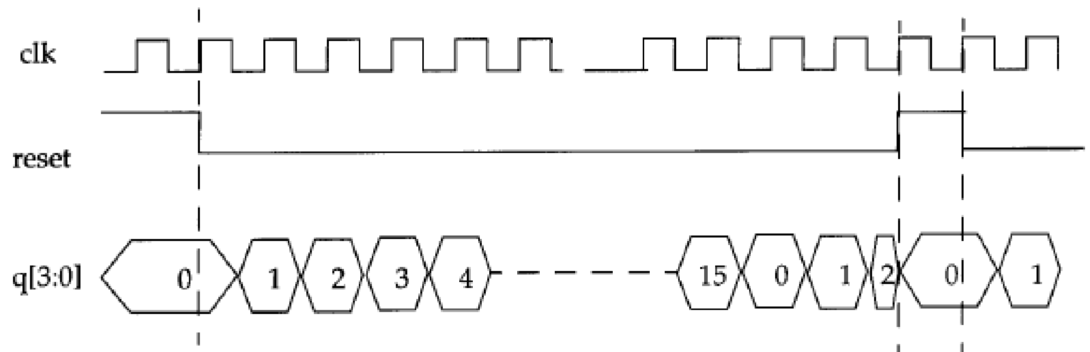


Fig. 3.2 Formele de unda ale semnalelor de intrare si iesire (*waveforms*)

Iesirea blocului de simulare:

```
  0 Output q =  0
 20 Output q =  1
 30 Output q =  2
 40 Output q =  3
 50 Output q =  4
 60 Output q =  5
 70 Output q =  6
 80 Output q =  7
 90 Output q =  8
100 Output q =  9
110 Output q = 10
120 Output q = 11
130 Output q = 12
140 Output q = 13
150 Output q = 14
160 Output q = 15
170 Output q =  0
180 Output q =  1
190 Output q =  2
195 Output q =  0
210 Output q =  1
220 Output q =  2
```

<u>Porturi</u>

Reprezinta interfata prin care un modul comunica cu lumea exterioara.

Exemplu:
Porturile unui circuit integrat sunt pinii de intrare/iesire ai acestuia.

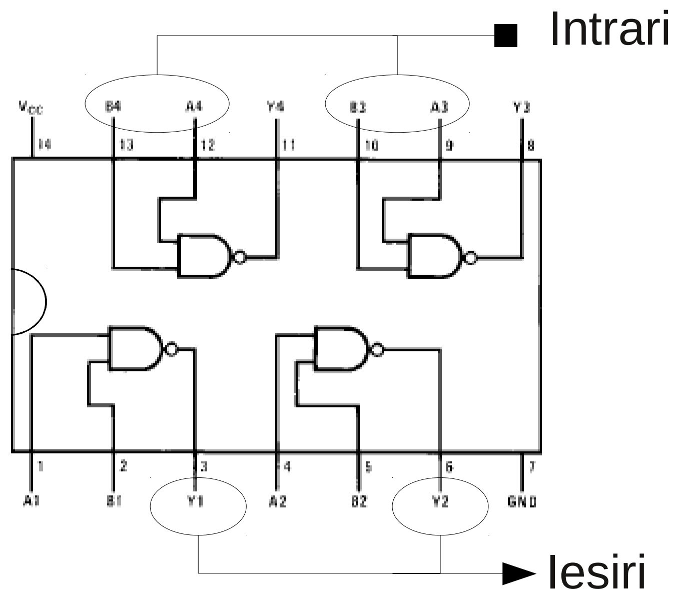


Fig. 4.1 Schema circuitului 7400 (4 porti NAND cu doua intrari)

Ca si in cazul unui circuit integrat, realizarea interna a modulului Verilog nu este vizibila din exterior, astfel ca proiectantului i se ofera o flexibilitate sporita.

Un modul poate sa nu aiba intrari sau iesiri.

Exemplu:

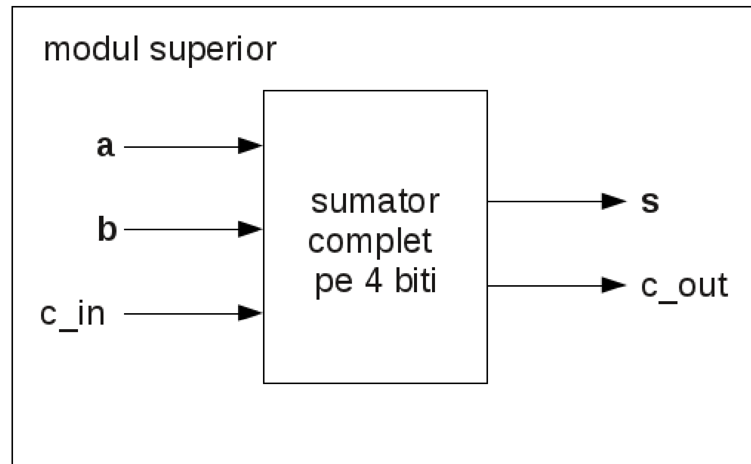


Fig. 4.2 Porturile de intrare-iesire pentru un sumator complet pe 4 biti

Modulul *sumator complet* din figura 4.2 efectueaza adunarea variabilelor **a**, **b** si c_in, punand rezultatul in variabila **s** si c_out. Modulul *superior* nu dispune de interfete cu exteriorul. Porturile, daca sunt prezente sunt enumerate intr-o lista de porturi:

```
module sumator_complet4bit(s,c_out,a,b,c_in);
module top;
```

=>Declararea porturilor

Este OBLIGATORIU ca toate porturile prezente in lista de porturi sa fie declarate expicit in interiorul modulului.

Cuvant cheie Verilog	Tip de port
input	port de intrare
output	port de iesire
inout	port bidirectional

Exemplu:

Modulul sumator complet pe 4 biti declarat mai devreme va avea urmatoarele declaratii de porturi:

```
module sumc4bit(s,c_out,a,b,c_in);
    output[3:0] s;
    output c_out;

    input [3:0] a,b;
    input c_in;

.   .   .   .   .   .   .   .   .

endmodule
```

Implicit, toate porturile declarate sunt de tipul “wire”. Daca dorim ca o iesire sa memoreze o valoare anume, va trebui sa o declaram explicit “reg”.

Exemplu:

```
module dff(q,d,clk,reset);
    output q;
    reg q;
    input d, clk, reset;
.   .   .   .   .   .   .   .
endmodule
```

Porturile declarate *input* si *inout* nu pot fi declarate si *reg*, deoarece acestea nu trebuie sa memoreze o valoare ci sa reflecte schimbarile aparute pe semnalele conectate la intrarie respective.

=> Reguli de conectare a porturilor

Porturi *input* – intern sunt intotdeauna de tipul fir, in exterior se pot conecta la o variabila care poate fi *reg* sau *fir* (eng. *net*).

Porturi *output* – intern pot fi atat de tip fir cat si *reg*, dar extern sunt intotdeauna conectate la un fir (nu se pot conecta la o variabila *reg*).

Porturi *inout* – intern sunt intotdeauna de tip fir si extern trebuie “legate” tot la variabile fir.

Porturi *neconectate* – limbajul Verilog accepta ca porturile sa ramana neconectate atunci cand sunt utilizate de exemplu pentru depanare.

Conectarea porturilor de intrare sau iesire de dimensiuni diferite sunt acceptate, dar se genereaza un avertisment.

Exemplu:

```
sumc4bit sum(s, ,a,b,c_in);//portul c_out nu a fost conectat
```

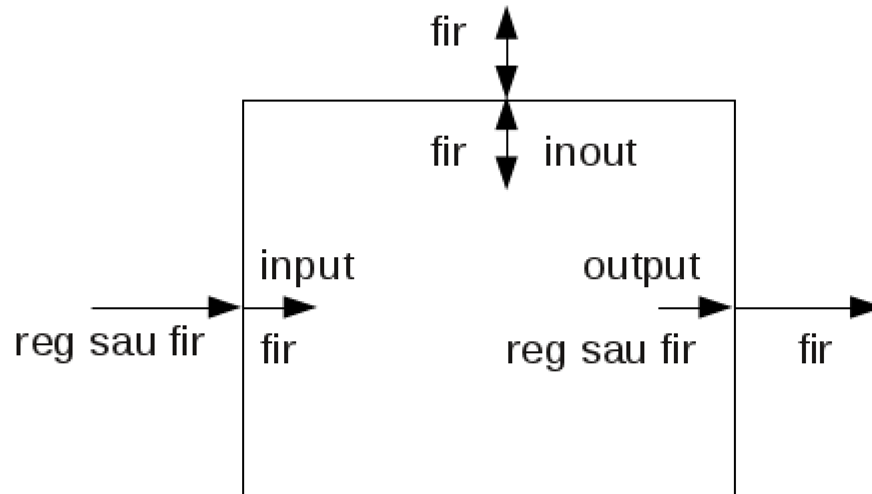


Fig. 4.3 Reguli de conectare a porturilor

Exercitiu:

In codul Verilog de mai jos sunt erori la declararea porturilor. Care declaratie nu este conforma cu regulile enuntate?

```
module top;
   reg [3:0] a,b;
   reg c_in;
   reg [3:0] s;
   wire c_out;

   sumc4bit sum0(s,c_out,a,b,c_in);
.  .  .  .  .  .  .  .  .  .  .
endmodule
```

Raspuns:

```
module top;
    reg [3:0] a,b;
    reg c_in;
    reg [3:0] s;//iesirea trebuie declarata aici ca “wire”
    wire c_out;

    sumc4bit sum0(s,c_out,a,b,c_in);
.   .   .   .   .   .   .   .   .   .   .
endmodule
```

Conectarea porturilor la semnale externe

=> Conectarea prin lista ordonata – este mai intuitiva, dar necesita cunoasterea ordinii exacte a porturilor din definitia modulului.

Exemplu:

```
module top;
   reg[3:0] A,B;
   reg C_IN;
   wire[3:0] S;
   wire C_OUT;
   sumc4bit sum(S,C_OUT,A,B,C_IN);
.   .   .   .   .   .   .   .   .   .
endmodule
module sumc4bit(s,c_out,a,b,c_in);
   output[3:0] s;
   output c_out;
   input[3:0] a,b;
   input c_in;
.   .   .   .   .   .   .   .   .   .
endmodule
```

=> Conectarea porturilor prin nume – se utilizeaza atunci cand numarul porturilor a devenit destul de mare incat memorarea ordinii porturilor din definitia modulului este dificila. Ordinea poate fi aleatorie atat timp cat se cunosc numele porturilor definite.

Exemplu:

```
sumc4bit sum(.c_out(C_OUT),.s(S),.b(B),.c_in(C_IN),.a(A));
```

Nume ierarhice

Orice instanta a unui modul, semnal sau variabila este definita cu ajutorul unui *identificator*. Un identificator oarecare poate fi unic localizat in ierarhia proiectului.

Un *nume ierarhic* este format dintr-o lista de identificatori separati prin “.” pentru fiecare nivel ierarhic, astfel incat orice identificator poate fi adresat din orice loc prin specificarea numelui ierarhic complet.

Modulul superior este numit si modul radacina deoarece nu este instantiat nicaieri. Pentru a atribui un nume unic unui identificator, se incepe de la modulul superior si se parcurge “drumul” catre identificator prin toate nivelele ierarhice.

Exemplu:

```
module bistRS(q,nq,ns,nr);
    output q,nq;
    input ns,nr;

    nand n1(q,ns,nq);
    nand n2(nq,nr,q);
endmodule
module top;
    wire Q,nQ;
    reg nS,nR;
    bistRS bist(Q,nQ,nS,nR);
    initial
        begin
            nS=0;nR=0;
        #5 nR=1;
        #5 nR=0;
        #5 nS=1;
        end
endmodule
```

Exemplu de atribuire ierarhica a numelor:

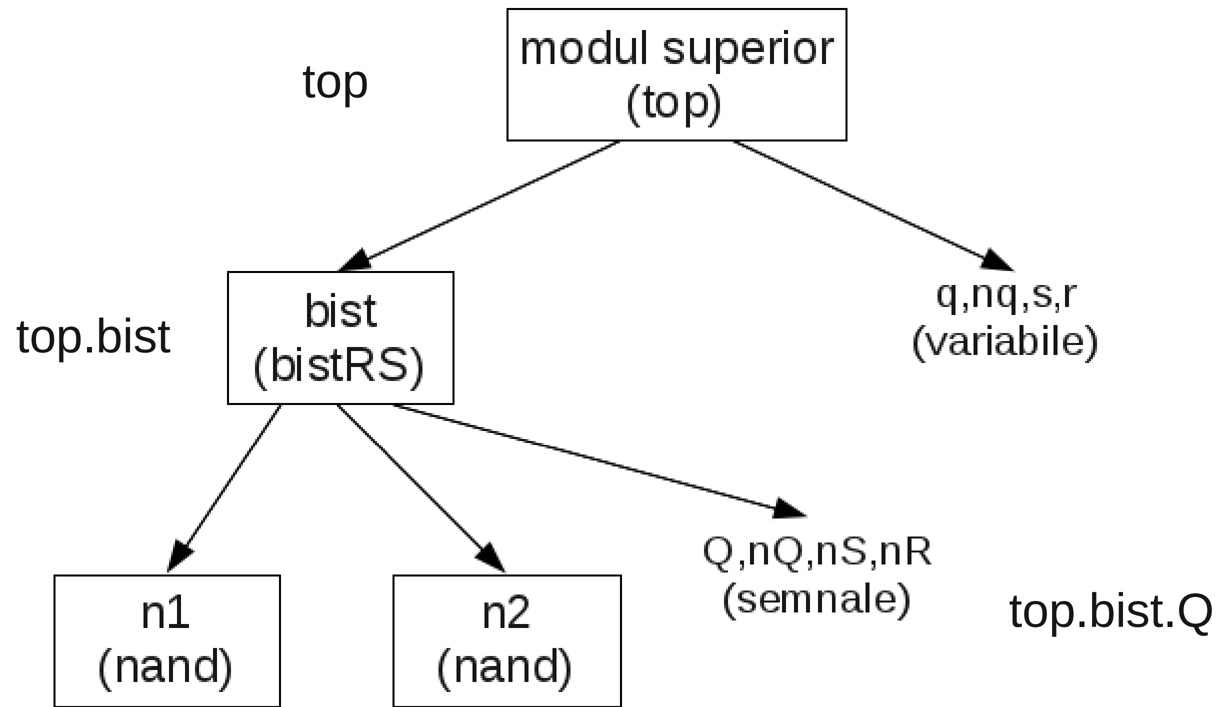


Fig. 4.4 Atribuire ierarhica a numelor

Intrebari

1. Care sunt componentele de baza ale unui modul? Care dintre aceste componente sunt obligatorii?

2. Un modul care nu interactioneaza cu alte module are vreun port de intrare/iesire? In definitia modulului trebuie sa apara o lista de porturi?

5. Modelarea in Verilog cu ajutorul portilor logice

5.1 Tipuri de porti logice

Portile logice formeaza *primitive* in Verilog, care sunt instantiate asemanator cu orice modul, nemaifiind necesara definirea lor. Exista doua tipuri de porti logice: *and/or* si *buf/not*.

=> Porti de tip *and/or*

- au o singura iesire si multiple intrari
- primul termen din lista porturilor este iesirea, restul fiind intrari
- iesirea unei porti este re-evaluata imediat ce una din intrari s-a modificat
- portile *and/or* disponibile sunt: *and*, *or*, *xor*, *nand*, *nor*, *xnor*

Exemplu:

```
wire out,in1,in2,in3;  //declarare intrari si iesiri
and a1(out,in1,in2);   //instantiere de porti logice
nand na1(out,in1,in2);
nor nor1(out,in1,in2);
//instantiere de poarta logica cu 3 intrari
nand na1_3(out,in1,in2,in3);
//instantiere fara precizarea numelui portii
and(out,in1,in2);
```

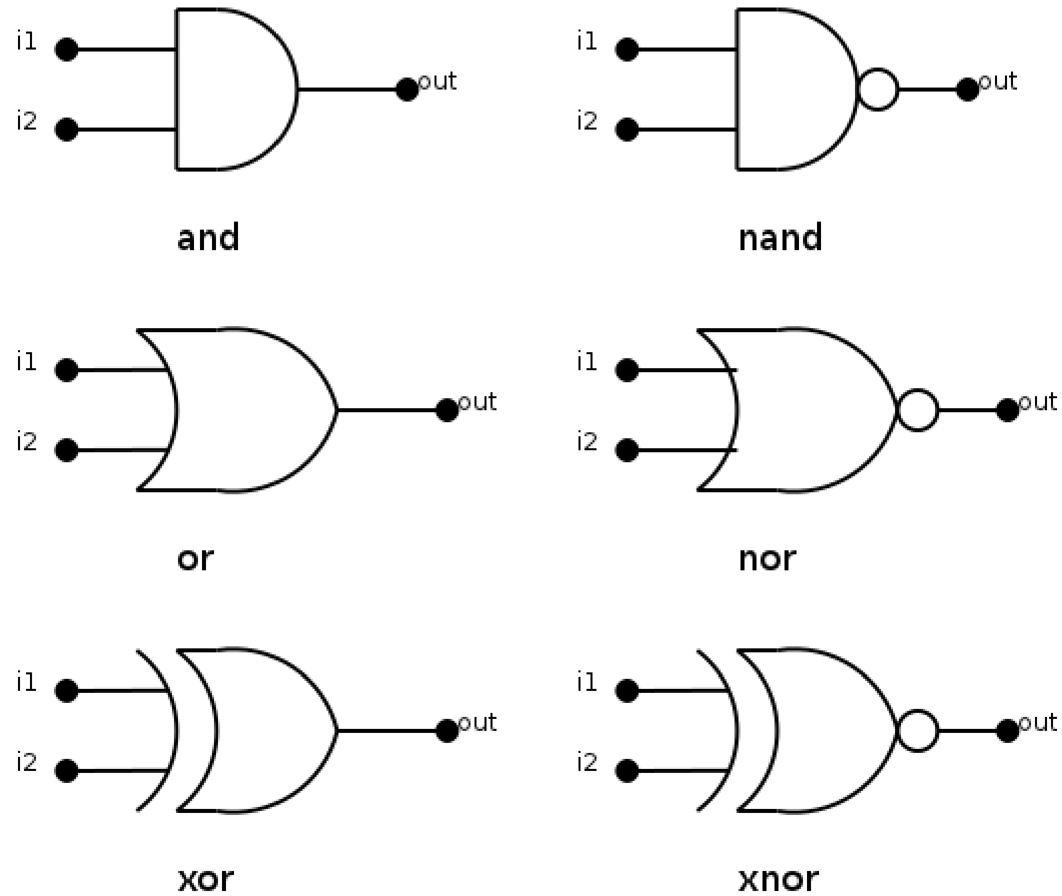


Fig. 5.1 Portile logice de baza

Tabele de adevar pentru portile de tip *and*/*or*:

and		i1			
		0	1	x	z
	0	0	0	0	0
	1	0	1	x	x
i2	x	0	x	x	x
	z	0	x	x	x

or		i1			
		0	1	x	z
	0	0	1	x	x
	1	1	1	1	1
i2	x	x	1	x	x
	z	x	1	x	x

xor		i1			
		0	1	x	z
	0	0	1	x	x
	1	1	0	x	x
i2	x	x	x	x	x
	z	x	x	x	x

nand		i1			
		0	1	x	z
	0	1	1	1	1
	1	1	0	x	x
i2	x	1	x	x	x
	z	1	x	x	x

nor		i1			
		0	1	x	z
	0	1	0	x	x
	1	0	0	0	0
i2	x	x	0	x	x
	z	x	0	x	x

xnor		i1			
		0	1	x	z
	0	1	0	x	x
	1	0	1	x	x
i2	x	x	x	x	x
	z	x	x	x	x

=> Porti de tip *buf/not*

- au una sau mai multe iesiri, dar o singura intrare
- ultimul port din lista de porturi e conectat la intrare
- portile primitive disponibile in Verilog sunt: *buf* si *not*

Simbolurile folosite in schemele logice pentru cele doua tipuri de porti sunt:

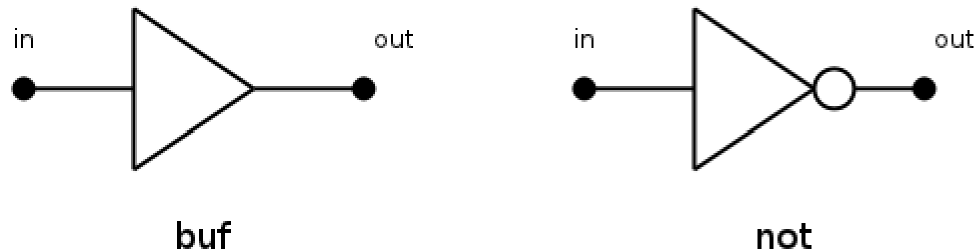


Fig. 5.2 Portile de tip *buf/not*

Exemplu:

```
buf b1(out1,in);
not n1(out2,in);
buf b1_2(out1,out2,in); //poarta cu mai multe iesiri
not(out1,in);    //instantiere fara precizarea numelui portii
```

Tabele de adevar pentru portile de tip *buf/not*:

buf in	out
0	0
1	1
x	x
z	x

not in	out
0	1
1	0
x	x
z	x

In Verilog sunt disponibile porti de tip *buf/not* cu intrare de comanda: *bufif1, bufif0, notif1, notif0*. Semnalele prin aceste porti se transmit numai daca intrarea de comanda este activata, altfel la iesire apare **z**:

Exemplu:

```
bufif1 b1(out,in,ctrl);
bufif0 b0(out,in,ctrl);
notif1 n1(out,in,ctrl);
notif0 n0(out,in,ctrl);
```

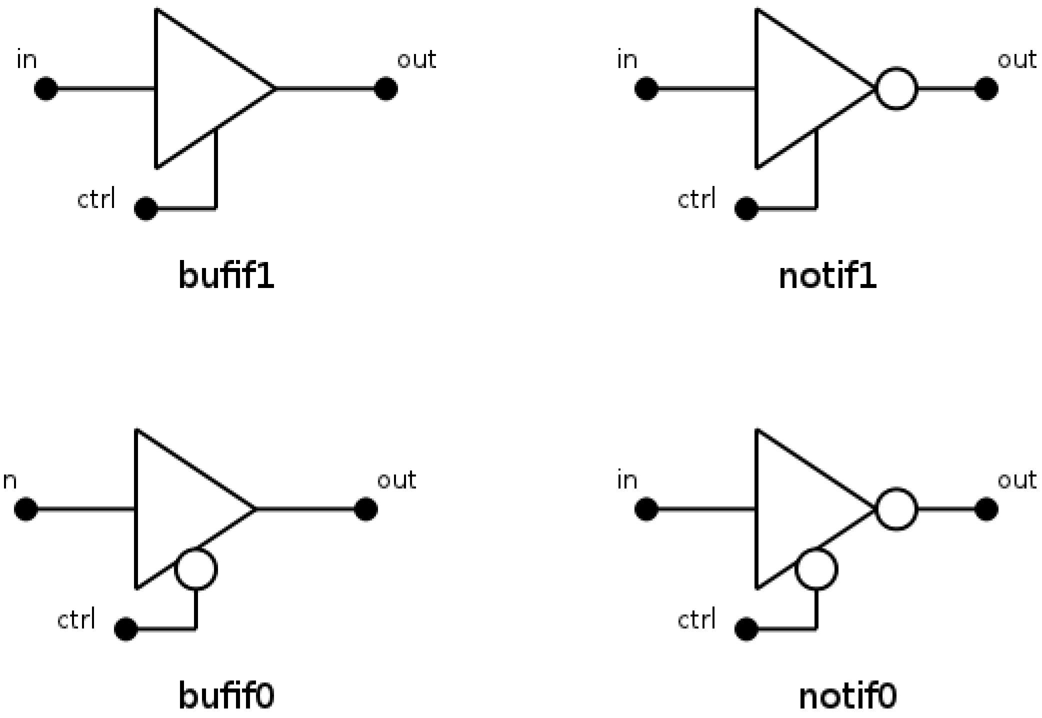


Fig. 5.3 Portile de tip bufif/notif

Exemplu complet – sumatorul pe 4 biti

s = a XOR b XOR c_in
c_out = (a AND b) OR c_in AND (a XOR b)

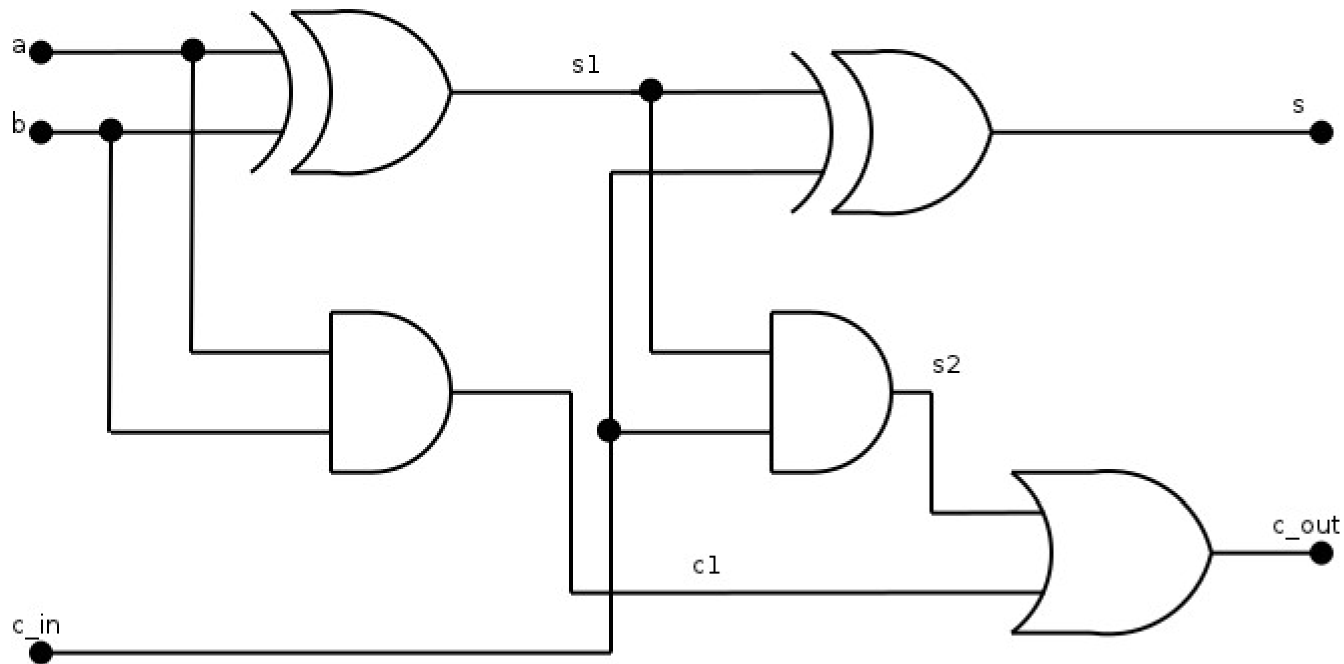


Fig. 5.4 Sumator complet pe 1 bit

```
module sum1bit(s,c_out,a,b,c_in);
    input  a,b,c_in; //declaratii porturi intrare/iesire
    output s,c_out;
    wire   s1,s2,c1; //declaratii conexiuni interne
    xor(s1,a,b);
    and(c1,a,b);
    xor(s,s1,c_in);
    and(c2,s1,c_in);
    or(c_out,c2,c1);
endmodule
```

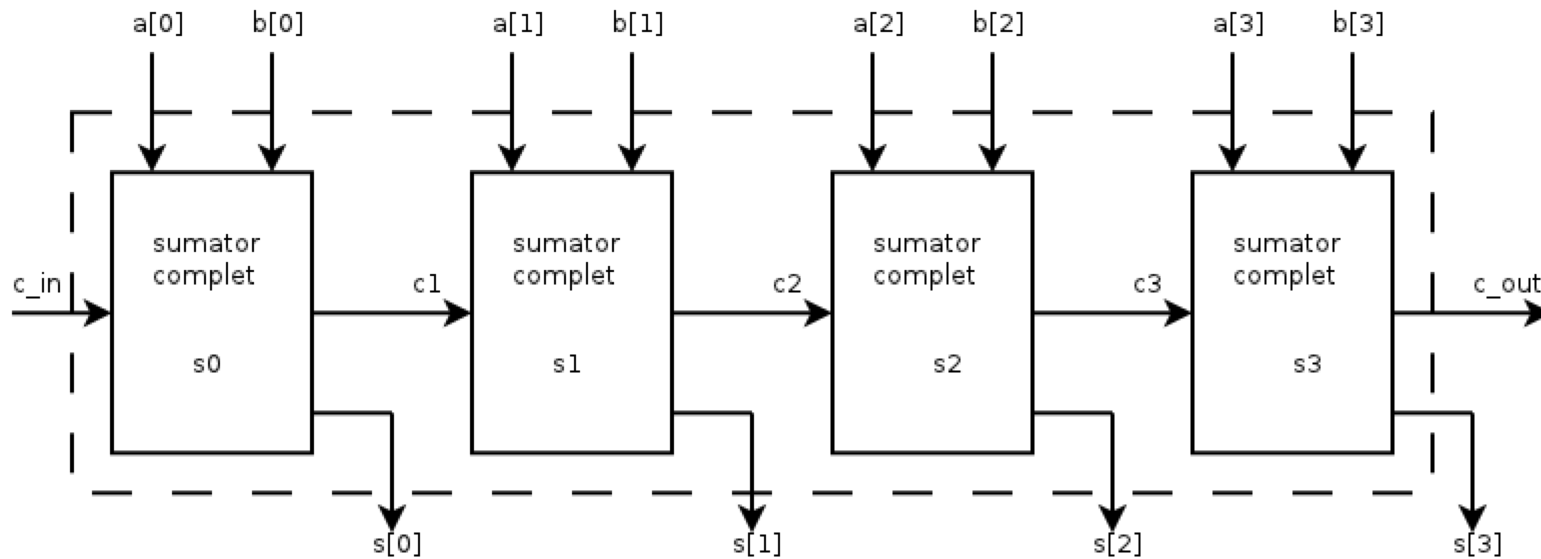


Fig. 5.5 Sumator complet pe 4 biti

```
module sum4bit(s,c_out,a,b,c_in);
    input[3:0] a,b;  //declaratii porturi de intrare/iesire
    input c_in;
    output[3:0]s;
    output c_out;
    wire c1,c2,c3;   //declaratii conexiuni interne
    //instantieri sumatoare complete pe 1 bit
    sum1bit s0(s[0],c1,a[0],b[0],c_in);
    sum1bit s1(s[1],c2,a[1],b[1],c1);
    sum1bit s2(s[2],c3,a[2],b[2],c2);
    sum1bit s3(s[3],c_out,a[3],b[3],c3);
endmodule

module simulare;
    reg[3:0]A,B; //declaratii variabile
    reg C_IN;
    wire[3:0]S;
    wire C_OUT;
    sum4bit s4bit(S,C_OUT,A,B,C_IN);
    initial
        begin
        $monitor($time,” A=%b B=%b C_IN=%b => S=%b C_OUT=
                %b”,A,B,C_IN,S,C_OUT);
        end
```

```
    initial
    begin
            A=4'd0;B=4'd0;C_IN=1'b0;
        #5 A=4'd3;B=4'd4;
        #5 A=4'd2;B=4'd5;
        #5 A=4'd9;B=4'd9;
        #5 A=4'd1;B=4'd15;
        #5 A=4'd10;B=4'd5;C_IN=1'b1;
    end
endmodule
```

```
0 A= 0000, B=0000, C_IN= 0, --- C_OUT= 0, SUM= 0000

5 A= 0011, B=0100, C_IN= 0, --- C_OUT= 0, SUM= 0111

10 A= 0010, B=0101, C_IN= 0, --- C_OUT= 0, SUM= 0111

15 A= 1001, B=1001, C_IN= 0, --- C_OUT= 1, SUM= 0010

20 A= 1010, B=1111, C_IN= 0, --- C_OUT= 1, SUM= 1001

25 A= 1010, B=0101, C_IN= 1,, C_OUT= 1, SUM= 0000
```

Fig. 5.6 Rezultatul simularii

5.2 Intarzieri asociate portilor logice

In circuitele reale, semnalele nu se transmit instantaneu, pentru a modela cat mai fidel transmisia semnalului printr-o poarta logica fizica, in Verilog se pot specifica *intarzieri* (eng. delay).

=> Intarziere pe frontul pozitiv – este asociata cu tranzitia de la un nivel logic oarecare (0,x sau z) la 1.

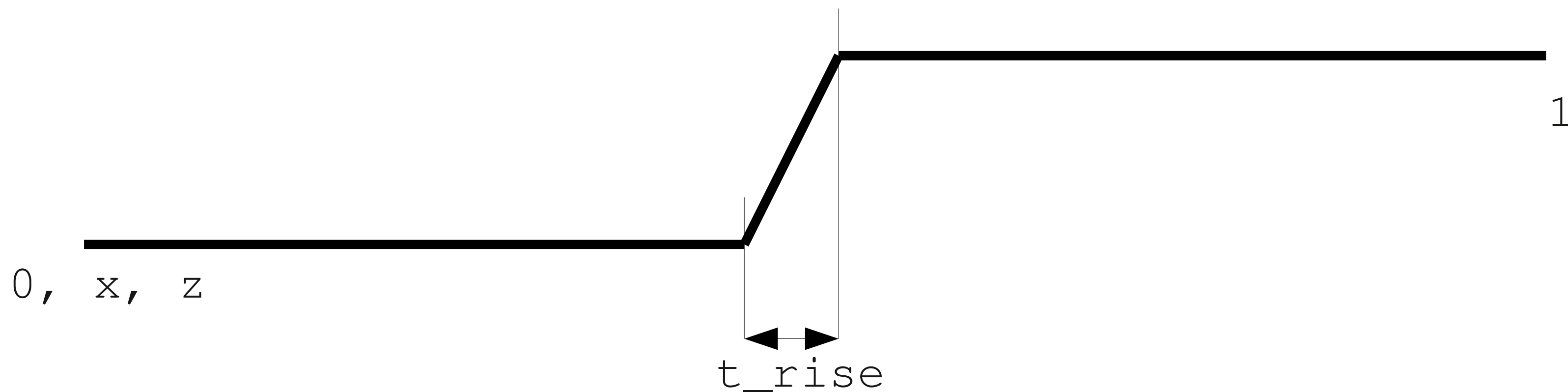


Fig. 5.7 Tranzitie pozitiva

=> Intarziere pe frontul negativ – este asociata cu tranzitia la nivelul logic 0 de la un nivel logic oarecare (1, x sau z).

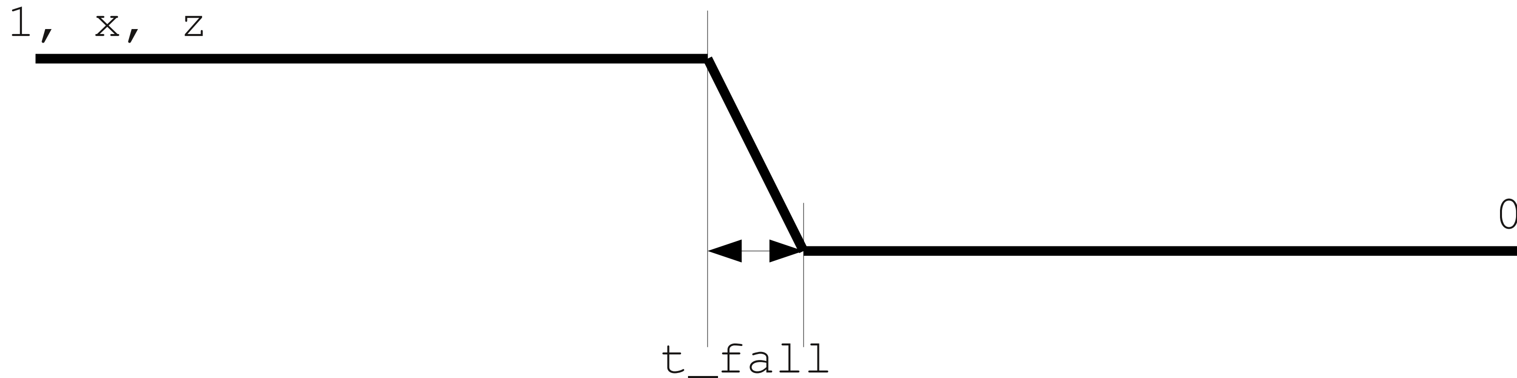


Fig. 5.8 Tranzitie negativa

=> Intarziere la tranzitia in starea **z –** numita si intarziere de blocare (eng. *turn-off delay*)
=> Intarziere la tranzitia in starea **x** se considera cea mai mica dintre cele trei.

Reguli de asociere a intarzierilor:

- daca se specifica o singura intarziere, aceasta e valabila pentru toate tranzitiile.
- daca se specifica doua intarzieri, acestea sunt asociate cu frontul pozitiv si negativ, cel de blocare fiind egal cu cel mai mic dintre ele.
- daca se specifica trei intarzieri, acestea vor fi referite ca front pozitiv, negativ si blocare.

Exemplu:

```
//tipuri de intarzieri
//specificarea unei singure valori pentru toate intarzierile
and #(delay_time) a1(out,i1,i2);

//specificarea intarzierilor pentru front pozitiv si negativ
and #(rise_t,fall_t) a2(out,i1,i2);

//specificarea tuturor intarzierilor
bufif0 #(rise_t,fall_t,turn_off) b1(out,in,control);

//************************************************************
//valori numerice pentru intarzieri
and #(5) a1(out,i1,i2);
and #(4,6) a2(out,i1,i2);
bufif0 #(4,5,6) b(out,in,control);
```

Exemplu de aplicare practica:

Se considera un modul numit *func* ce calculeaza urmatoarea functie logica:

$$out = (a \cdot b) + c$$

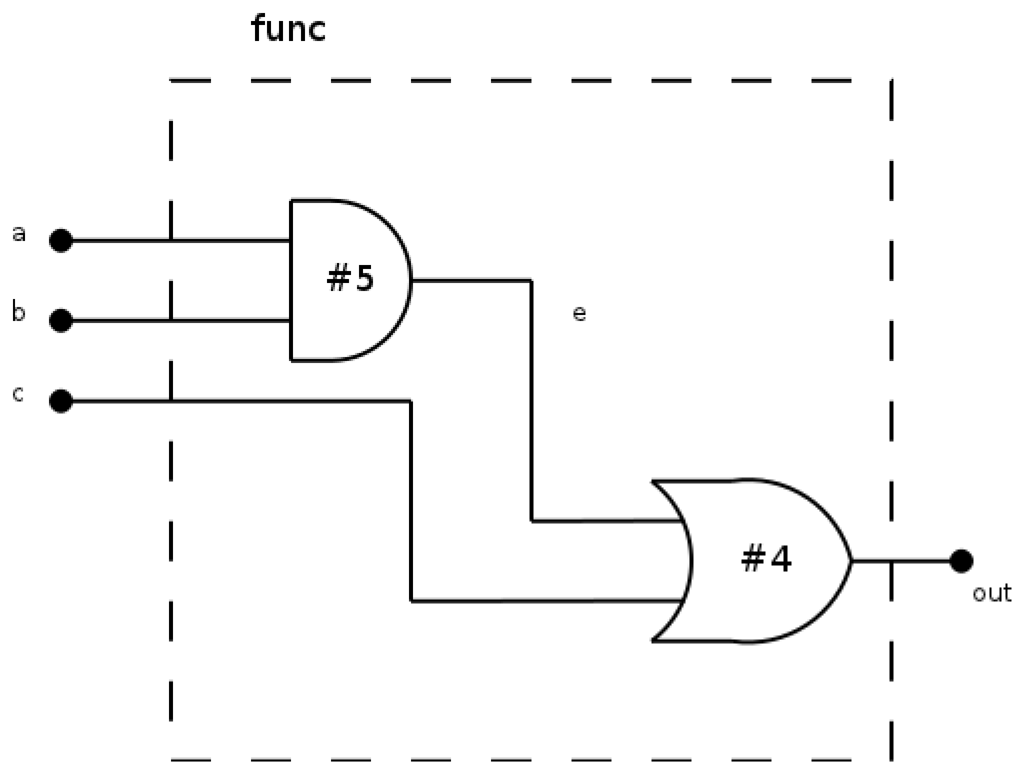


Fig. 5.9 Schema logica a functiei *func*

Modulul *func* este definit in continuare:

```
module func(out,a,b,c);
    output out;
    input a,b,c;
    wire e;
    and #(5) p1(e,a,b);
    or #(4) p2(out,e,c);
endmodule

module stimulus;
    reg A,B,C;
    wire O;
    func f1(O,A,B,C);
    initial
        begin
            $dumpfile("func.vcd");
            $dumpvars();
            A=1'b0; B=1'b0; C=1'b0;
        #10 A=1'b1; B=1'b1; C=1'b1;
        #10 A=1'b1; B=1'b0; C=1'b0;
        #20 $finish;
        end
endmodule
```

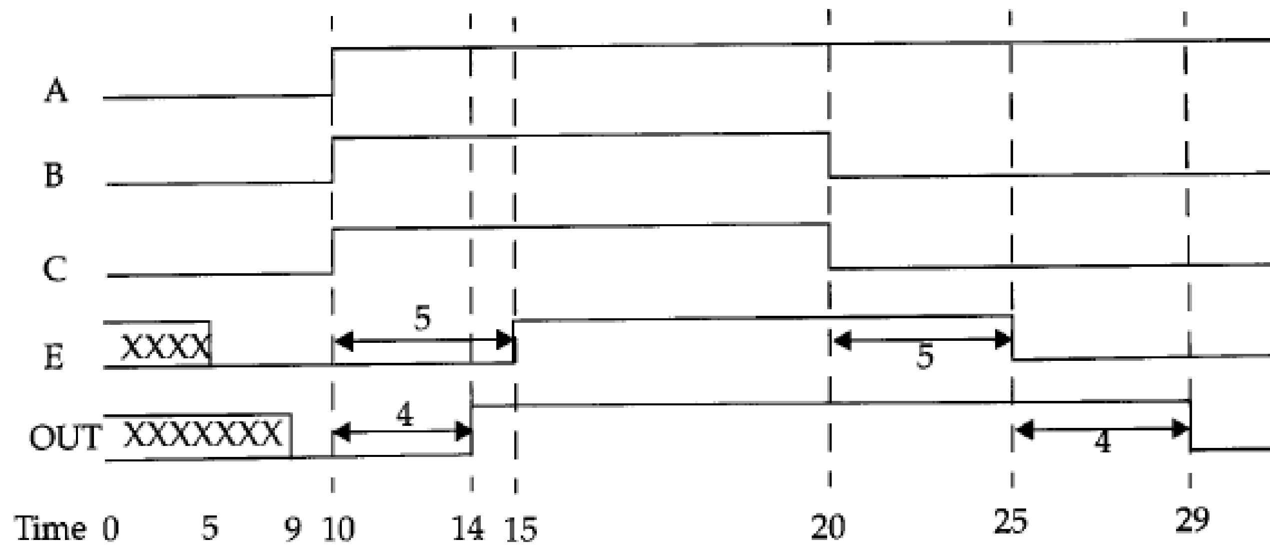


Fig. 6.0 Formele de unda ale marimilor din modulul *func*

6. Modelarea RTL(eng. *Register Transfer Level*) in Verilog

Circuitele digitale complexe nu mai pot fi descrise util in termeni de porti logice; pentru aceasta este preferata descrierea prin transferul intre registri: RTL. La randul sau, aceasta modalitate de descriere se divide in modelare prin flux de date (eng. *dataflow modeling*) si modelare prin comportament (eng. *behavioral modeling*).

Verilog permite modelarea RTL, modelare intens folosita pentru proiectarea circuitelor moderne, foarte complexe, care utilizeaza din plin instrumentele de sinteza logica.

6.1 Asignarea continua

Este instructiunea de baza in modelarea RTL, folosita pentru a modifica incontinuu valoarea logica aflata pe un fir. Inlocuieste portile logice in descrierea circuitului, fiind o reprezentare mai abstracta.

Reguli pentru asignare continua:

1. Membrul stang al unei asignari continue este intotdeauna un scalar sau vector de tip fir, nu poate fi de tip reg.
2. Asignarile continue sunt tot timpul active: expresia din membrul drept al asignarii este re-evaluata imediat ce unul din operanzii implicati se modifica, dupa care este atribuita membrului stang.
3. Se pot specifica intarzieri sub forma unui numar oarecare de unitati de timp. Aceasta intarziere reprezinta timpul dupa care expresia din membrul drept este atribuita membrului din stanga.

Exemplu:

```
//asignarea operatiei logice SI intre i1 si i2 variabilei out
assign out = i1 & i2;
//asignare pentru variabile de tip vector de reg si fir
assign addr[15:0] = addr1_bits[15:0] ^ addr2_bits[15:0];
//asignare in urma unei operatii de concatenare
assign {c_out,sum[3:0]} = a[3:0] + b[3:0] + c_in;
```

=> Asignarea continua implicita se face la declarare

Exemplu:

```
wire out;
assign out =in1 & in2;  //asignare continua

wire out = in1 & in2;   //asignare continua implicita
```

6.2 Intarzieri asociate cu o asignare

Prima metoda specifica o intarziere printr-un numar oarecare de unitati de timp in interiorul liniei unde se apeleaza *assign*.

Exemplu:

assign #10 out = in1 & in2;

Orice modificare a valorii variabilelor i1 sau i2 va fi intarziata cu 10 unitati de timp pana la re-evaluarea expresiei in1 & in2.

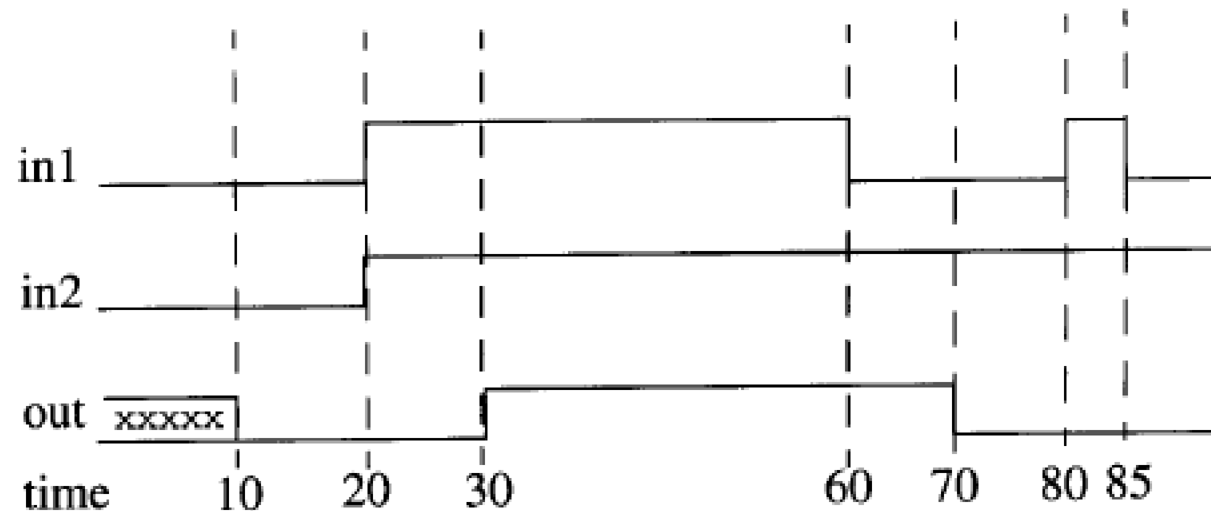


Fig. 6.1 Intarzieri asociate cu asignarea

- daca aceste valori se modifica intre timp (in timpul celor 10 unitati), se vor considera valorile din momentul calcularii.
- un impuls aplicat pe intrari cu durata mai mica decat intarizerea de propagare nu se transmite la iesire.

A doua metoda specifica o intarziere pe un fir direct la declarare, astfel ca orice modificare ulterioara in valoarea asignata este intarziata.

Exemplu:

```
wire #10 out;
assign out = in1 & in2;
//codul de mai sus are acelasi efect cu urmatorul
wire out;
assign #10 out = in1 & in2;
```

6.3 Expresii, operatori, operanzi

Tipuri de operatori	Simbol	Operatie	Numar de operanzi
Aritmetici	*	inmultire	2
	/	impartire	2
	+	adunare	2
	-	scadere	2
	%	modulo	2
Logici	!	negatie logica	1
	&&	SI logic	2
	||	SAU logic	2
Relationali	>	mai mare ca	2
	<	mai mic ca	2
	>=	mai mare sau egal cu	2
	<=	mai mic sau egal cu	2
Egalitate	==	egalitate	2
	!=	inegalitate	2
	===	egalitate generala	2
	!==	inegalitate generala	2

<table>
<tr><th>Tipuri de operatori</th><th>Simbol</th><th>Operatie</th><th>Numar de operanzi</th></tr>
<tr><td>Pe bit</td><td>~</td><td>negatie pe bit</td><td>1</td></tr>
<tr><td></td><td>&</td><td>SI pe bit</td><td>2</td></tr>
<tr><td></td><td>|</td><td>SAU pe bit</td><td>2</td></tr>
<tr><td></td><td>^</td><td>XOR pe bit</td><td>2</td></tr>
<tr><td></td><td>^~ sau ~^</td><td>XNOR pe bit</td><td>2</td></tr>
<tr><td>De contragere</td><td>&</td><td>SI</td><td>1</td></tr>
<tr><td></td><td>~&</td><td>SI-NU</td><td>1</td></tr>
<tr><td></td><td>|</td><td>SAU</td><td>1</td></tr>
<tr><td></td><td>~|</td><td>SAU-NU</td><td>1</td></tr>
<tr><td></td><td>^</td><td>XOR</td><td>1</td></tr>
<tr><td></td><td>^~ sau ~^</td><td>XNOR</td><td>1</td></tr>
<tr><td>De deplasare</td><td>>></td><td>Deplasare dreapta</td><td>2</td></tr>
<tr><td></td><td><<</td><td>Deplasare stanga</td><td>2</td></tr>
<tr><td>Concatenare</td><td>{ }</td><td>Concatenare</td><td>oricat</td></tr>
<tr><td>Copiere</td><td>{ { } }</td><td>Copiere</td><td>oricat</td></tr>
<tr><td>Conditional</td><td>?:</td><td>Conditie</td><td>3</td></tr>
</table>

=> Operatori de contragere

```
//x = 4'b1010
&x //este echivalent cu 1 & 0 & 1 & 0 => rezultat 1'b0
|x //este echivalent cu 1 | 0 | 1 |  0  => rezultat 1'b1
^x //este echivalent cu 1 ^ 0 ^ 1 ^ 0   => rezultat 1'b0
```

=> Operatorul de concatenare

```
//a=1'b1, b=2'b00, c=2'b10, d=3'b110

y={b,c}              //rezultatul este 4'b0010
y={a,b,c,d,3'b001}   //rezultatul este 11'b10010110001
y={a,b[0],c[1]}      //rezultatul este 3'b101
```

=> Operatorul de copiere

```
reg a;
reg[1:0] b,c;
reg[2:0]d;
a=1'b1; b=2'b00; c=2'b10; d=3'b110;
Y={ 4{a} };          //rezultatul este 4'b1111
Y={ 4{a}, 2{b} };    //rezultatul este 8'b11110000
Y={ 4{a},2{b},c };   //rezulatul este 10'b1111000010
```

=> Operatorul conditional

<conditie> ? <expresie_adeavarat> : <expresie_fals>

Mai intai se evalueaza *conditie*, in caz ca este evaluata adevarat, se executa *expresie_adevarat*, altfel se executa *expresie_fals*.

Exemplu:

```
//implementarea unui multiplexor 2:1
assign out = comanda ? in1 : in 0;
```

6.4 Modelarea algoritmica

Proiectantii de calculatoare stabilesc o arhitectura finala pentru o anumita aplicatie dupa ce au fost evaluate mai multe alternative. Modul de a gandi solutia unei probleme nu implica detalierea pana la nivelul portii logice, ci stabilirea unui *algoritm* pe care apoi il vor implementa in hardware. Numai dupa finalizarea acestei etape se poate trece la implementarea fizica a circuitului rezultat.

Verilog ofera posibilitatea descrierii *comportamentului* (eng. *behavior*) unui circuit, utilizand o reprezentare mult abstractizata. La acest nivel de reprezentare, descrierea circuitului este asemanatoare limbajului structurat C, desi nu trebuie confundata reprezentarea unui circuit hardware si programarea procedurala.

=> Proceduri structurate

In Verilog exista doua astfel de proceduri: *initial* si *always*. Aceste doua instructiuni sunt fundamentale pentru reprezentarea algoritmica, toate celelealte constructii algoritmice pot aparea NUMAI in interiorul celor doua proceduri.

Verilog este un limbaj de programare *concurentiala*, spre deosebire de limbajul C care este prin definitie un limbaj *secvential*.

Fluxurile separate de instructiuni ruleaza in paralel; fiecare procedura *initial* sau *always* reprezinta un flux separat care “porneste” la momentul de timp 0.

! Instructiunile *initial* si *always* nu pot fi imbricate.

Instructiunea *initial*

Un bloc *initial* este executat o singura data, la inceputul simularii, la momentul de timp 0, dupa care nu mai este executat. Daca exista mai multe blocuri *initial*, acestea sunt executate incepand cu momentul 0, independent unul de celalalt si terminarea acestora este, de asemenea independenta. Daca sunt prezente mai multe instructiuni, acestea se grupeaza folosind cuvintele cheie *begin* si *end*.

Exemplu:

```
module stimulus;
reg x,y,a,b,m;
initial
   m=1'b0;       //o singura instructiune
initial
   begin
#5  a=1'b1;      //instructiuni multiple
#25 b=1'b0;
   end
initial
   begin
#10 x=1'b0;
#25 y=1'b1;
   end
initial
#50 $finish;
endmodule
```

Executia instructiunilor aflate in interiorul blocurilor *initial* se va face in urmatoare ordine:

Timp	Instructiune executata
0	`m=1'b0`
5	`a=1'b1`
10	`x=1'b0`
30	`b=1'b0`
35	`y=1'b1`
50	`$finish`

Blocurile initial se folosesc pentru initializare, monitorizare, forme de unda sau alte prelucrari ce se realizeaza o singura data la inceputul rularii.

Instructiunea *always*

Instructiunile aflate intr-un bloc *always* se executa intr-o iteratie (bucla) continua incepand de la momentul de timp 0.

Exemplu:

```
module clk_gen;
   reg clk;

   initial
      clk=1'b0;
   always         //semnalul de ceas este inversat la fiecare
                  //semiperioada (perioada=2x10=20)
      #10 clk=~clk;
   initial
      #100 $finish;
endmodule
```

Se observa ca initializarea semnalului de ceas se face intr-un bloc *initial* separat. De asemenea, intr-un alt bloc *initial* se termina executia simularii prin apelarea $finish (sau $stop). Absenta instructiunii de terminare a executiei duce la continuarea simularii pentru un timp teoretic infinit.

Asignari procedurale

Acest tip de asignare modifica valoarea unei variabile de tip *reg*, *integer*, *real* sau *time*. Valoarea asignata la un moment dat ramane valida atat timp cat nu este asignata o alta valoare, spre deosebire de *asignarea continua* (o modificare a expresiei aflate in membrul drept conducea automat la o modificare a membrului stang).

Membrul stang al unei asignari procedurale poate fi:

- o variabila reg, integer, real, time sau o memorie
- un bit selectat al unei variabile din tipurile enumerate (ex. addr[0])
- o selectie a unei variabile (ex. addr[31:16])
- o concatenare a variabilelor expuse mai sus

=>Asignari procedurale *blocante*

Sunt executate in ordinea in care se gasesc in blocul secvential respectiv. O astfel de asignare blocheaza celelalte asignari care ii urmeaza.

Folosesc simbolul “=”.

Exemplu:

```
reg x,y,z;
reg[15:0] reg_a,reg_b;
integer count;
initial
    begin
        x=0;y=1;z=1;
        count=0;
        reg_a=16'b0;reg_b=reg_a;
    #15 reg_a[2]=1'b1;
    #10 reg_b[15:13]={x,y,z};
        count=count+1;
    end
```

Instructiunea y=1 se executa numai dupa ce x=0; instructiunile de asignare blocante consecutive sunt executate secvential.

- instructiunile de la x=0 la reg_b=reg_a se executa la timpul 0;
- reg_a[2]=1'b1 la momentul de timp=15;
- reg_b[15:13]={x,y,z} la momentul de timp=25;
- instructiunea count=count+1 la timpul 25.

=> <u>Asignari procedurale neblocante</u>

Asignarile neblocante permit programarea asignarilor fara a bloca executia instructiunilor ce urmeaza in blocul secvential respectiv. Se foloseste operatorul “<=”, este acelasi simbol cu “mai mic sau egal”.

Exemplu:

```
reg x,y,z;
reg[15:0] reg_a,reg_b;
integer count;

initial
    begin
        x=0;y=1;z=1;
        count=0;
        reg_a=16'd0;reg_b=reg_a;
        reg_a[2]<= #15 1'b1;
        reg_b[15:13]<= #10 {x,y,z};
        count<=count+1;
    end
```

Instructiunile de la x=0 pana la reg_b=reg_a sunt executate secvential la momentul de timp t=0, dupa care cele trei asignari neblocante sunt executate in acelasi moment de timp:

- reg_a[2]=1'b1 este *programata* sa se execute dupa 15 unitati de timp;
- reg_b[15:13]={x,y,z} este programata sa se execute dupa 10 unitati de timp;
- count=count+1 este programata sa fie executat fara intarziere.

<u>Aplicatii ale atribuirilor neblocante</u>

1. Modelarea mai multor transferuri de date ce au loc dupa un anumit eveniment.

```
always @(posedge clk)
begin
   reg1 <= #1 in1;
   reg2 <= @(negedge clk) in2 ^ in3;
   reg3 <= #1 reg1; //asignarea vechii valori a lui reg1
end
```

La fiecare aparitie a unui front pozitiv au loc urmatoarele evenimente:

a. se efectueaza o operatie de citire pentru variabilele aflate in membrul drept al fiecarei atribuiri: in1, in2, in3 si reg1;

b. operatiile de scriere in variabilele aflate in membrul stang sunt programate pentru a fi executate la momentele de timp specificate prin intarzieri: scriere in reg1 dupa o unitate de timp, scriere in reg2 la urmatorul front negativ si in reg3 dupa o unitate de timp;

c. operatiile de scriere sunt executate dupa programul stabilit. Ordinea in care se efectueaza scrierea NU este importanta, deoarece se utilizeaza valori stocate intern. De exemplu, in registrul reg3 se stocheaza vechea valoare a lui reg1, valoare ce a fost stocata intern la pasul a., chiar daca in reg1 a fost stocata o alta valoare inainte ca operatia de scriere in reg3 sa fi avut loc.

2. Interschimbarea a doi registri la fiecare front pozitiv de ceas

```
//ex.1 doua secvente concurente cu asignari blocante
always @(posedge clk)
   a=b;
always @(posedge clk)
   b=a;

//ex.2 doua secvente concurente cu asignari neblocante
always @(posedge clk)
   a<=b;
always @(posedge clk)
   b<=a;
```

In exemplul 1 apare o conditie de tip “race” (eng. cursa, nedeterminare). Se va executa mai intai a=b sau b=a in functie de implementarea specifica a simulatorului, astfel ca cele doua valori nu vor fi interschimbate, ci la sfarsitul executiei, cei doi registri vor avea aceeasi valoare.

Exemplul 2 elimina aceasta conditie de nedeterminare prin folosirea asignarilor neblocante: la aparitia unui front pozitiv, valorile variabilelor aflate in membrul drept sunt citite si stocate in *variabile temporare*. Aceste valori sunt apoi asignate membrului stang. Prin separarea operatiei de scriere de cea de citire se asigura interschimbarea corecta a celor doua variabile indiferent de ordinea in care se fac operatiile de scriere.

Folosirea asignarilor neblocante este indicata acolo unde apar transferuri concurentiale de date in urma aparitiei unui eveniment comun. Asignarile blocante pot produce aparitia conditiilor de nedeterminare, din cauza faptului ca rezultatul final este dependent de ordinea in care s-au facut asignarile, fapt eliminat in cazul asignarilor neblocante datorita independentei de ordinea in care sunt evaluate asignarile.

Exemple de domenii de aplicare sunt modelarea arhitecturilor pipeline sau a transferurilor de date exclusiv mutuale.

Dezavantajele sunt reprezentate de un consum mai mare de resurse ale simulatorului.

Modalitati de control al timpului in Verilog

Lipsa structurilor de control a timpului in Verilog opreste desfasurarea simularii. Exista trei metode de control al timpului:

- bazata pe specificarea unei intarzieri

- bazata pe aparitia unui eveniment

- bazat pe aparitia unui nivel logic

1. Metoda bazata pe specificare unei intarzieri

Presupune specificarea unui numar de unitati de timp de intarziere intre incarcarea unei instructiuni si executia ei. Exista mai multe metode de a specifica aceasta intarziere, dar toate folosesc notatia #:

a. Intarzierea obisnuita este marcata de prezenta unui numar de unitati de timp la stanga instructiunii

Exemplu

```
//definirea unor constante
parameter latenta=20;
parameter delta=2;
//definirea unor variabile de tip reg
reg x,y,z,p,q;
initial
   begin
         x=0;
      #10 y=1;//este  intarziata  asignarea  y=1  cu  10  unitati
            //de timp
      #latenta  z=0;
      #(latenta+delta) p=1;
      #y  x=x+1;
   end
```

In cadrul unei secvente *begin-end*, intarzierea este relativa la timpul de simulare curent (valoarea timpului in momentul “intalnirii” instructiunii), astfel ca instructiunea z=0 se executa dupa un timp de 10+latenta=10+20=30 unitati de timp.

b. Intarzierea specificata in interiorul unei asignari

Specificarea unei intarzieri la stanga instructiunii este inlocuita in acest caz prin specificare in dreapta operatorului “=”.

Exemplu

```
//definirea unor variabile de tip registru
reg x,y,z;
//intarziere specificata in interiorul unei asignari
initial
   begin
   x=0;z=0;
   y= #5 x + z; //se citesc valorile x si z la momentul 0, se
                //evalueaza x+z si se asteapta 5 unitati de
                //timp pana la asignarea variabilei y
   end
```

```
//cod echivalent in care folosim intarzieri obisnuite
initial
   begin
      x=0;z=0;
      temp_xz=x+z; //se citesc valorile lui x si z la
                   //momentul de timp curent si se
                   //stocheaza intr-o variabila temporara
   #5 y=temp_xz;   //Desi x si z se pot schimba intre timp,
                   //valoarea asignata lui y ramane
                   //neschimbata
   end
```

=>Intarzierea bazata pe aparitia unui eveniment

Evenimentul este definit ca schimbarea valorii aflate intr-un registru sau o variabila fir (eng. *net*). Evenimentele pot fi folosite la declansarea executiei unei instructiuni sau a unei secvente de instructiuni.

Exista patru tipuri de blocuri de comanda (eng. *control*) a intarzierilor bazate pe eveniment:

a) bloc de comanda normal (eng. *regular*)

Foloseste simbolul “@” pentru a specifica un bloc de comanda normal. Instructiunile se executa la schimbarea valorii unui semnal sau la aparitia unui front pozitiv/negativ (`posedge/negedge`).

Exemplu:

```
@(clk) q=d;   //instructiunea q=d se executa de cate ori
             //semnalul 'clk' isi schimba valoarea
@(posedge clk) q=d; //q=d se executa atunci cand semnalul
                    //'clk' are o tranzitie pozitiva
                    //(0->1, x sau z, x->1, z->1)
@(negedge clk) q=d; //q=d se executa atunci cand semnalul
                    //'clk' are o tranzitie negativa
                    //(1->0, x sau z, x->0, z->0)
```

b) bloc de comanda nominal (eng. *named event*)

Limbajul Verilog permite declararea unui tip eveniment. Pentru a declara un eveniment se foloseste cuvantul cheie *event*. Declansarea evenimentului se face cu ajutorul simbolului “->”, iar identificarea cu simbolul “@”.

Exemplu:

```
event rec_data;      //declararea unui eveniment 'rec_data'
always @(posedge clk)
begin
   if(last_data)    //daca s-a receptionat ultimul pachet
      -> rec_data   //se declanseaza evenimentul 'rec_data'
end

always @(rec_data)  //se asteapta declansarea 'rec_data'
                    //la aparitia evenimentului sunt stocate
                    //doua pachete de date intr-un buffer
data_buf={data_pkt[0],data_pkt[1]};
```

c) bloc de comanda SAU

In anumite situatii este necesara declansarea executiei unei instructiuni sau secvente de instructiuni la aparitia unei tranzitii pe oricare semnal din mai multe semnale. O asemenea situatie este modelata cu o functie logica SAU (se foloseste cuvantul cheie 'or').

Exemplu:
```
//bistabil activ pe nivel (nu pe front)
always @(reset or clk or d)   //se asteapta modificarea
                 //valorii semnalelor 'reset', 'clk' sau 'd'
begin
   if(reset)     //daca 'reset' se afla in '1' logic, iesirea
      q=1'b0;    //'q' ia valoarea 0
   else
      if(clk)    //daca semnalul 'clk' este in '1' logic,
      q=d;       //intrarea este memorata si transmisa la
                 //iesirea 'q'
end
```

d) bloc de comanda pe nivel

In acest caz se asteapta ca o conditie specificata sa fie adevarata pentru a putea executa o instructiune sau o secventa de instructiuni. Pentru modelarea acestui tip de comanda se foloseste cuvantul cheie *wait*.

Exemplu:

```
always
   wait (enable_count) #30 count=count+1;
```

Nivelul logic al semnalului enable_count este continuu monitorizat. Daca nivelul se pastreaza '0', instructiunea nu este executata, insa daca enable_count devine 1, atunci instructiunea count=count+1 se executa dupa 20 de unitati de timp.

=> Instructiuni conditionale

Instructiunile conditionale se utilizeaza atunci cand executia unei instructiuni depinde de indeplinirea unor conditii (se folosesc cuvintele cheie *if else*). Exista trei tipuri de constructii *if-else*:

```
if(<expresie>) instructiune_adevarat;    //1. instructiune_adevarat se executa daca <expresie>
                                         //este adevarata

if(<expresie>) instructiune_adevarat;    //2. daca <expresie> este falsa se executa
                                         //instructiune_fals
else instructiune_fals;

if(<expresie1>) instructiune_adevarat1;        //3. Conditii 'if-else' imbricate, numai o singura
else if (<expresie2>) instructiune_adevarat2;//instructiune se va executa
else if (<expresie3>) instructiune_adevarat3;
else instructiune_implicita;
```

Daca <expresie> este evaluata la valoarea 'x' sau 'z', atunci este considerata falsa (echivalenta cu 0). O valoare mai mare sau egala cu 1 pentru <expresie> se considera adevarata.

=> Instructiunea de selectie multipla

Este folosita atunci cand numarul de expresii evaluate este prea mare pentru a folosi *if-else*.

```
case(<expresie>)
    optiunea1:instructiunea1;
    optiunea2:instructiunea2;
              .
              .
    default:  instructiune_implicita;
endcase
```

Exemplu comparativ *if-else* si *case-endcase*:

```
//semnale asociate unei Unitati Aritmetico-Logice
//exemplul 1
if(ual_control==0)
   y=x+z;
else if(ual_control==1)
   y=x-z;
else if(ual_control==2)
   y=x*z;
else
   $display(“Semnal UAL invalid”);

//exemplul 2
reg[1:0] ual_control;
.  .  .
case(ual_control)
   2'd0:y=x+z;
   2'd1:y=x-z;
   2'd2:y=x*z;
   default: $display(“Semnal UAL invalid”);
endcase
```

Exemplu multiplexor 4:1

```
module mux41(out,i0,i1,i2,i3,s0,s1);
    output out;
    input i0,i1,i2,i3;
    input s0,s1;
    reg out;

    always @(s1 or s0 or i0 or i1 or i2 or i3)
    case({s1,s0})
        2'd0:out=i0;
        2'd1:out=i1;
        2'd2:out=i2;
        2'd3:out=i3;
        default: $display("Semnale de selectie invalide");
    endcase
endmodule
```

Exemplu demultiplexor 1:4 la care se iau in considerare si starile 'x' si 'z':

```
module dmux14(out0,out1,out2,out3,in,s1,s0);
    output out0,out1,out2,out3;
    reg out0,out1,out2,out3;
    input s0,s1;
    input in;

    always @(s1 or s0 or in)
    case({s1,s0})
        2'b00:begin out0=in;out1=1'bz;out2=1'bz;out3=1'bz;end
        2'b01:begin out0=1'bz;out1=in;out2=1'bz;out3=1'bz;end
        2'b10:begin out0=1'bz;out1=1'bz;out2=in;out3=1'bz;end
        2'b11:begin out0=1'bz;out1=1'bz;out2=1'bz;out3=in;end

        2'bx0,2'bx1,2'bxz,2'bxx,2'b0x,2'b1x,2'bzx:
        begin
            out0=1'bx;out1=1'bx;out2=1'bx;out3=1'bx;
        end
        2'bz0,2'bz1,2'bzz,2'b0z,2'b1z:
        begin
            out0=1'bz;out1=1'bz;out2=1'bz;out3=1'bz;
        end
            default: $display("Semnale de selectie invalide");
        endcase
endmodule
```

Instructiunile 'casex' si 'casez'

- casex considera toate valorile 'x' sau 'z' ca indiferente
- casez considera toate valorile 'z' ca indiferente

Exemplu de selectare a unei stari intr-o masina de stare finita (eng. *Finite State Machine*), in care doar un bit se considera semnificativ, ceilalti sunt ignorati (se foloseste 'casex'):

```
reg[3:0] codare;
integer stare;

casex(codare)
   4'b1xxx:stare_urm=3;
   4'bx1xx:stare_urm=2;
   4'bxx1x:stare_urm=1;
   4'bxxx1:stare_urm=0;
   default: stare_urm=0;
endcase
```

=> Structuri iterative

Exista patru constructii iterative in Verilog: *while*, *for*, *repeat* si *forever*. Sintaxa lor este asemanatoare limbajului C. Instructiunile iterative pot aparea numai in interiorul structurilor *always* sau *initial*.

Iteratia “while”

Executa o secventa de instructiuni pana cand conditia specificata devine falsa. O conditie falsa de la inceput nu permite nicio executie a secventei.

Exemplu:

```
    //Se incrementeaza variabila count de la 0 la 127, secventa se
termina la valoarea 128,      //variabila fiind afisata la fiecare
pas.
    integer count;

    initial
        begin
            count=0;
            while(count<128)
                begin
                    $display(“count=%d”,count);
                    count=count+1;
                end
        end
```

Exemplul 2:

```
//Este cautata prima valoare logica '1' a unui bit dintr-un vector
//numit 'flags'
    `define TRUE 1'b1;
    `define FALSE 1'b0;
    reg[15:0] flags;
    integer i;
    reg continue;

    initial
    begin
            flag=16'b0010_0000_0000_0000;
            i=0;
            continue=`TRUE;
            while((i<16)&&(continue))
            begin
                if(flag[i])
                    begin
                $display(“S-a gasit un bit TRUE la pozitia: %d”,i);
                        continue=`FALSE;
                    end
            i=i+1;
            end
    end
```

Instructiunea iterativa 'for'

Este alcatuita din trei sectiuni in care se foloseste o variabila contor:

- conditie initiala
- verificare a indeplinirii conditiei de continuare
- o asignare procedurala pentru modificarea variabilei contor

Exemplu:

```
//Initializarea unui vector
`define MAX_STATES 32
integer state [0:`MAX_STATES-1];//sir de 32 de variabile
                                //'integer' (0:31)
integer i;

initial
   begin
for(i=0;i<32;i=i+2) //se initializeaza locatiile pare cu '0'
         state[i]=0;
for(i=1;i<32;i=i+2)//se initializeaza locatiile impare cu '1'
         state[i]=1;
   end
```

Instructiunea iterativa 'repeat'

Se utilizeaza atunci cand o secventa se executa de un numar exact de ori. Nu se poate folosi cu o conditie logica generala. Numarul de repetari al iteratiei este specificat de la inceput si nu se poate modifica pe parcursul executiei acesteia.

Exemplu:

```
//Variabila 'count' este incrementata de la 0 la 127 si
//valoarea ei afisata la fiecare pas
integer count;

initial
begin
    count=0;
    repeat(128)
        begin
            $display("count=%d",count);
            count=count+1;
        end
end
```

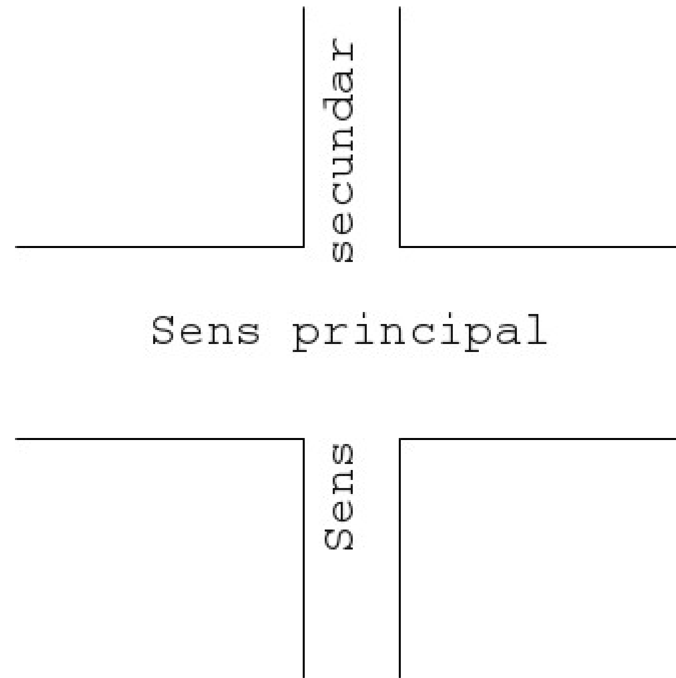
Instructiunea iterativa 'forever'

Aceasta instructiune nu contine nicio expresie care sa fie evaluata la fiecare iteratie. Se executa un timp nelimitat pana la intalnirea directivei *$finish*. Acest tip de iteratie este echivalent cu *while(1)*. Iteratia 'forever' se poate interupe cu directiva *disable*.

Exemplu:

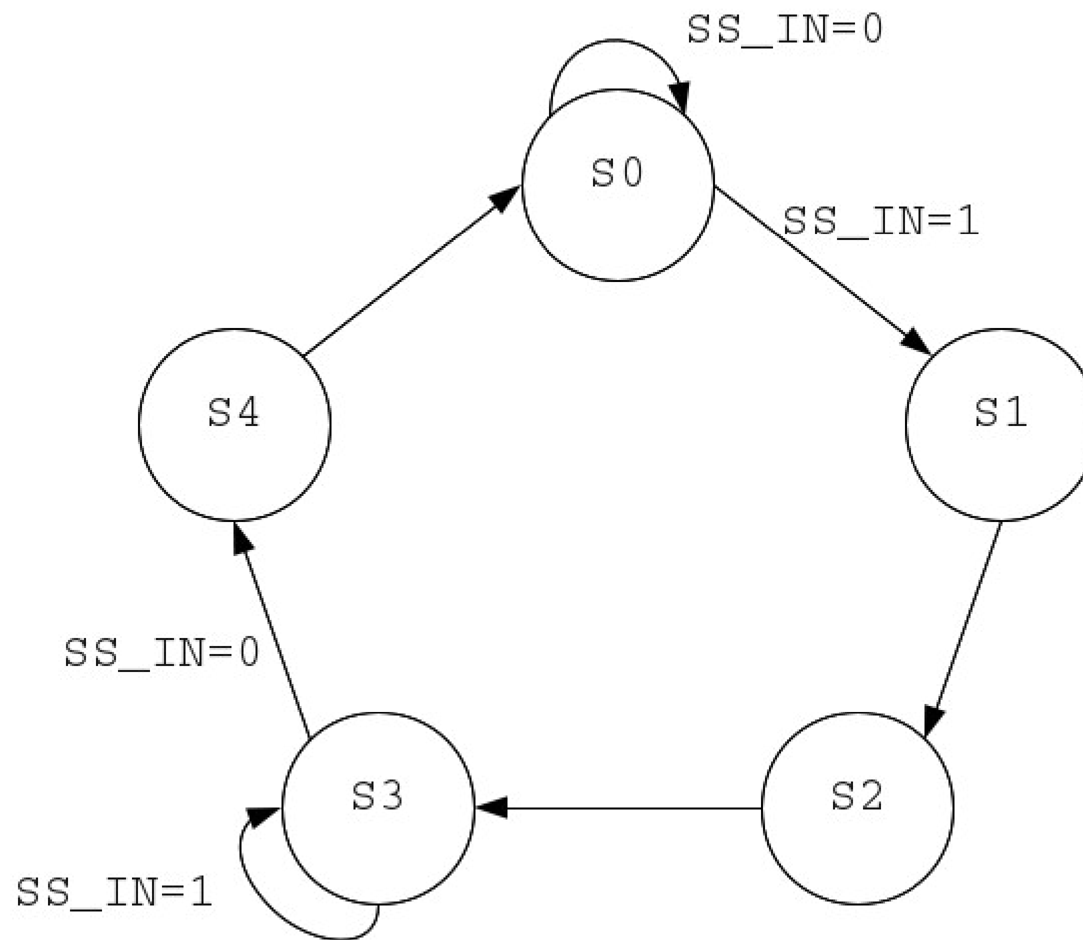
```
//Generarea unui semnal de ceas
reg clock;

initial
begin
   clock=1'b0;
   forever #10 clock=~clock;  //semnal de ceas cu perioada de
                              //20 unitati de timp
end
```

Exemplu Automat cu Stari Finite (eng. *Finite State Machine*) – controlul unei intersectii



Functionarea automatului

- exista doua semafoare: Sens Principal-SP si Sens Secundar-SS
- semnalele de trafic pentru sensul principal sunt prioritare fata de sensul secundar
- ocazional sosesc automobile pe sensul secundar. Semaforul devine verde doar atat cat este necesar ca acestea sa poata trece
- din momentul eliberarii sensului secundar, semaforul SS devine galben si apoi rosu, iar SP devine inapoi verde
- exista un senzor SS_IN care detecteaza prezenta automobilelor pe sensul secundar: SS_IN=1 daca sunt prezente automobile, altfel SS_IN=0



Stare	Iesiri	
S0	SP=G	SS=R
S1	SP=Y	SS=R
S2	SP=R	SS=R
S3	SP=R	SS=G
S4	SP=R	SS=Y

```
//modulul  pentru  implementarea  semaforului  cu  ajutorul  unui
//automat Moore
`define   TRUE   1'b1
`define   FALSE  1'b0
`define   R      2'd0
`define   Y      2'd1
`define   G      2'd2

//definirea starilor          SP      SS
`define   S0  3'd0      //GREEN      RED
`define   S1  3'd1      //YELLOW     RED
`define   S2  3'd2      //RED        RED
`define   S3  3'd3      //RED        GREEN
`define   S4  3'd4      //RED        YELLOW

//definirea intarzierilor
`define   Y2RDELAY  3   //Intarziere galben-rosu
`define   R2GDELAY  2   //Intarziere rosu-verde
```

```
module semaphore(SP,SS,SS_IN,clear,clk);
   output[1:0]SP,SS;   //iesirile au doi biti pentur a codifica
                       //cele trei culori
   reg[1:0]SP,SS;      //iesirile se declara si ca 'reg'
   input SS_IN;        //intrare pentru detectarea automob.pe SS
   input clk,clear;

   reg[2:0]state,next_state;//variabile pentru stari

   initial                 //initializare in starea S0
      begin
         state=`S0;
         next_state=`S0;
         SP=`G;
         SS=`R;
      end

   always @(posedge clk)   //starea se modifica doar la aparitia
      state=next_state;    //frontului pozitiv de ceas
```

```
//stabilirea semnalelor de iesire
   always @(state)
      case(state)
         `S0:begin
               SP=`G;
               SS=`R;
            end
         `S1:begin
               SP=`Y;
               SS=`R;
            end
         `S2:begin
               SP=`R;
               SS=`R;
            end
         `S3:begin
               SP=`R;
               SS=`G;
            end
         `S4:begin
               SP=`R;
               SS=`Y;
            end
      endcase
```

```
//implementarea automatului propriu-zis folosind o structura
   always @(state or clear or SS_IN)
   begin
      if(clear)
         next_state=`S0;
      else
      case(state)
         `S0:if(SS_IN)
               next_state=`S1;
            else
               next_state=`S0;
         `S1:begin    //apare o intarziere la tranzitia S1->S2
               repeat(`Y2RDELAY) @posedge(clk);
               next_state=`S2;
            end
         `S2:begin    //apare o intarziere la tranzitia S2->S3
               repeat(`R2GDELAY) @posedge(clk);
               next_state=`S3;
            end
         `S3:if(SS_IN)
               next_state=`S3;
            else
               next_state=`S4;
```

```
          `S4:begin    //apare o intarziere la tranzitia S4->S0
                repeat(`Y2RDELAY)@(posedge clk);
                next_state=`S0;
             end
          default:
             next_state=`S0;
       endcase
    end
endmodule
```

7. Taskuri si functii

Frecvent in proiectarea circuitelor apare situatia in care o secventa de cod este prezenta in mai multe locuri in program. Pentru a evita rescrierea codului, cele mai multe limbaje ofera posibilitatea crearii procedurilor sau subrutinelor.

In limbajul Verilog un proiect complex la nivel algoritmic poate fi de descompus in

=> *taskuri*

=> *functii*.

Taskurile au argumente de intrare, iesire si intrare/iesire. Functiile au doar argumente de intrare.

Diferenta intre taskuri si functii:

Functii	Taskuri
O functie poate apela o alta functie dar nu poate apela un alt task.	Un task poate apela un alt task sau o alta functie.
Functiile au un timp de executie egal cu 0.	Taskurile se pot executa intr-un timp diferit de zero.
Functiile nu pot contine intarzieri, evenimente sau instructiuni pentru timp.	Taskurile pot contine intarzieri, evenimente sau instructiuni pentru timp.
Functiile trebuie sa aiba cel putin un argument de intrare. Pot avea mai multe intrari.	Taskurile pot avea unul sau mai multe argumente de tipul *input*, *output* sau *inout*.
Functiile returneaza intotdeauna o singura valoare. Nu pot avea argumente de tipul *inout* sau *output*.	Taskurile nu returneaza o valoare, insa pot transfera multiple valori prin argumente de tipul *output* sau *inout*.

Atat taskurile cat si functiile sunt declarate in interiorul unui modul si sunt locale.

=> Functiile sunt utilizate atunci cand codul Verilog este pur combinational, se executa intr-un timp 0 si au un singur rezultat (ex.conversii)

=> Taskurile contin intarzieri, evenimente si pot avea mai multe iesiri.

Taskuri

Un task se declara folosind cuvintele cheie *task* si *endtask*. Se pot declara porturi de intrare, iesire sau intrare/iesire. Acestea sunt folosite pentru a furniza date taskului si pentru a prelua rezultatele executiei acestuia.

Exemplul 1:

```
//ex.1 Se defineste un modul "operation" care contine un task
//"bitwise_oper"
.   .   .
parameter delay = 10;
reg[15:0] A,B;
reg[15:0] AB_AND, AB_OR, AB_XOR;
```

```
    always @(A or B)    //de cate ori A sau B isi schimba valoarea
    begin
//se apeleaza taskul "bitwise_oper". Taskul are doua intrari: A, B
//si trei iesiri: AB_AND, AB_OR si AB_XOR. Argumentele se scriu in
//ordinea in care apar in declararea taskului
          bitwise_oper(AB_AND,AB_OR,AB_XOR,A,B);
    end

    //definirea taskului "bitwise_oper"
    task bitwise_oper;
       output[15:0] ab_and,ab_or,ab_xor;
       input[15:0] a,b;
       begin
          #delay ab_and=a&b;
          ab_or=a|b;
          ab_xor=a^b;
       end
    endtask
    .   .   .
endmodule
```

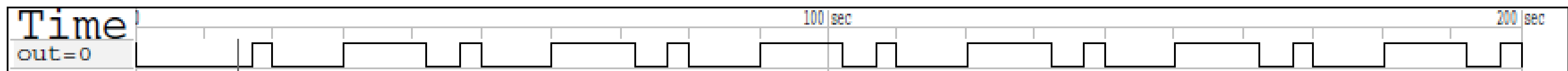
Cele trei iesiri sunt intarziate cu un interval de 10 unitati de timp.

Taskurile pot accesa variabile de tip “reg” definite in modul.

Exemplul 2:

```
//ex.2 definirea unui modul care genereaza un semnal de ceas
//asimetric
module sequence;
.   .   .
reg clock;
.   .   .
initial
    init_sequence;
.   .   .
always
    begin
        asymmetric_sequence;
    end
```

```
.   .   .
//definirea taskului de initializare
task init_sequence;
begin
   clock=1'b0;
end
endtask

//definirea taskului pentru generarea secventei asimetrice
task asymmetric_sequence;
   begin
      #12 clock=1'b0;
      #5  clock=1'b1;
      #3  clock=1'b0;
      #10 clock=1'b1;
   end
endtask
.   .   .
.   .   .
endmodule
```



Functii

Functiile se declara folosind cuvintele cheie *function* si *endfunction*. Particularitati ale functiilor Verilog:

- la declararea unei functii Verilog, implicit este declarata o variabila reg in interiorul modulului avand acelasi nume
- valoarea returnata de functie este inscrisa in variabila 'reg' la sfarsitul executiei functiei, aceasta valoare fiind plasata in locul apelarii functiei
- trebuie sa existe cel putin un argument al functiei

Exemplul 1:

```
//ex.1 Este modelat un circuit pentru calculul paritatii care
//returneaza o valoare pe 1 bit (scalara). Ca date de intrare
//se utilizeaza o adresa pe 32 biti. Paritatea este para.
module parity;
.   .   .
reg[31:0] addr;
reg parity;
```

```
    //se calculeaza o noua valoare a paritatii ori de cate ori se
    //modifica adresa
    always @(addr)
        begin
        parity=calc_parity(addr);  //apelarea functiei calc_parity
        $display("Paritate calculata=%b",calc_parity(addr));
                                    //a doua apelare a functiei
        end
    .   .   .
    .   .   .
    //definirea functiei de calculare a paritatii
    function calc_parity;
        input[31:0] address;
        begin
            //se utilizeaza registrul implicit cu acelasi nume
            //cu al functiei: calc_parity
            calc_parity=^address;//returneaza operatia SAU-EXCLUSIV
                                //asupra tuturor bitilor 'address'
        end
    endfunction
    .   .   .
    .   .   .
endmodule
```

La prima apelare a functiei 'calc_parity' valoarea returnata este stocata in variabila 'parity', iar la a doua apelare valoarea retunata inlocuieste apelul functiei.

Exemplul 2:

```
//ex2. Se defineste o functie care deplaseaza la stanga sau la dreapta cu un bit un registru pe //32 biti, in functie de un semnal de comanda.
//definirea modulului care contine functia 'shift'
Module shifter;
.  .  .
`define LEFT_SHIFT 1'b0
`define RIGHT_SHIFT 1'b1
reg[31:0] addr, left_addr, right_addr;
reg control;

//calculeaza valorile registrului 'addr' deplasate stanga sau
//dreapta ori de cate ori se modifica adresa
```

```
    always @(addr)
       begin
       //apelarea functiei 'shift' pentru a efectua deplasari
       //stanga sau dreapta
          left_addr=shift(addr,`LEFT_SHIFT);
          right_addr=shift(addr,`RIGHT_SHIFT);
       end
    .  .  .
    .  .  .
    //definirea functiei de deplasare. Iesire este o valoare pe 32
    //biti
    function[31:0] shift;
       input[31:0] address;
       input control;
       begin
    //valoarea returnata se calculeaza in functie de semnalul de
    //comanda
shift=(control==`LEFT_SHIFT) ? (address << 1) : (address >> 1);
       end
    endfunction
```

8. Modelarea cu elemente de comutare

Exista situatii practice in care este nevoie de a programa un circuit digital la cel mai scazut nivel, cel al tranzistorilor in regim de comutare (elemente de comutare).

Verilog nu pune la dispozitie implicit decat modele digitale pentru tranzistori, existand extensia “Verilog AMS” (Verilog Analog and Mixed Signal Extensions) pentru modelarea in regim analogic.

8.1 Elemente de modelare Verilog

=> <u>Tranzistori MOS</u>

Verilog modeleaza tranzistori de tip MOS de doua tipuri: *nmos* si *pmos*.
Cuvintele cheie Verilog pentru acesti tranzistori sunt:

```
nmos
pmos
```

Simbolurile celor doua tipuri de tranistori sunt prezentate in figura 8.1.

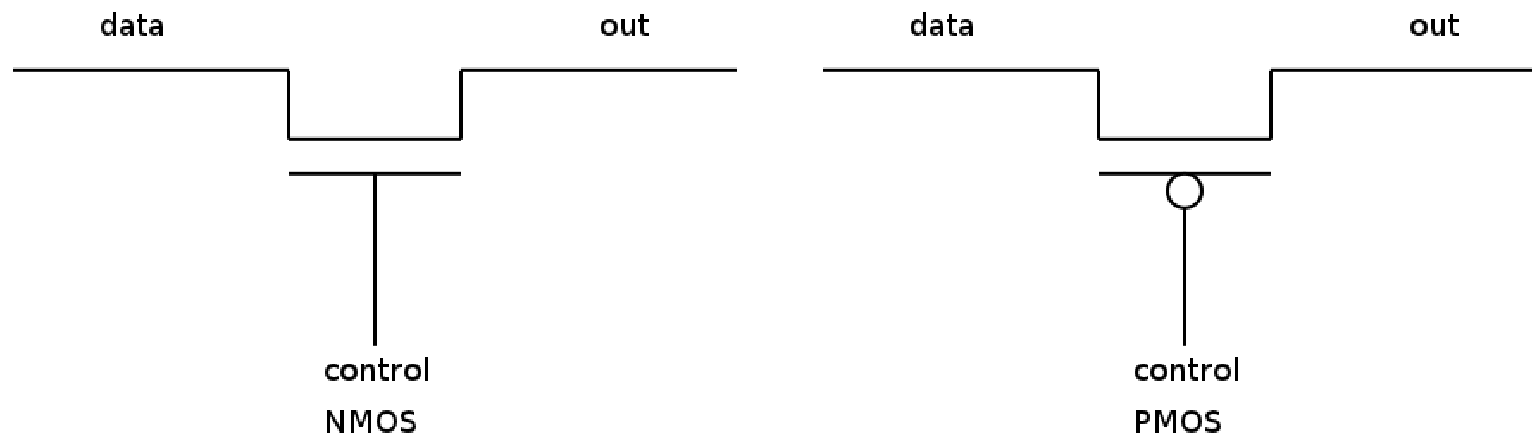


Figura 8.1 Tranzistori de tip NMOS si PMOS

Instantierea tranzistorilor se face in felul urmator:

```
nmos n1(out,data,control);//instantierea unui tranzistor nmos
pmos p1(out,data,control);//instantierea unui tranzistor pmos
```

La fel ca portile logice, numele atribuit tranzistorilor nmos si pmos este optional.

```
nmos (out,data,control);   //instantiere fara a preciza un nume
pmos (out,data,control);   //instantiere fara a preciza un nume
```

Valoarea iesirii este determinata de valoarea de pe intrarea 'data' si intrarea 'control'.

```
    nmos       control                 pmos          control
           0   1   x   z                              0   1   x   z
    ---------------------                     ---------------------
       0   z   0   L   L                         0    0   z   L   L

data   1   z   1   H   H          data           1    1   z   H   H

       x   z   x   x   x                         x    x   z   x   x

       z   z   z   z   z                         z    z   z   z   z
```

Valoarea H inseamna 1 sau z, iar L inseamna 0 sau z.

Tranzistorul nmos conduce daca intrarea de control se afla in 1, iar tranzistorul pmos conduce daca intrarea de control a acestuia este in 0.

=> <u>Elemente CMOS</u>

Elementele de comutare CMOS se declara cu cuvintul cheie *cmos*.

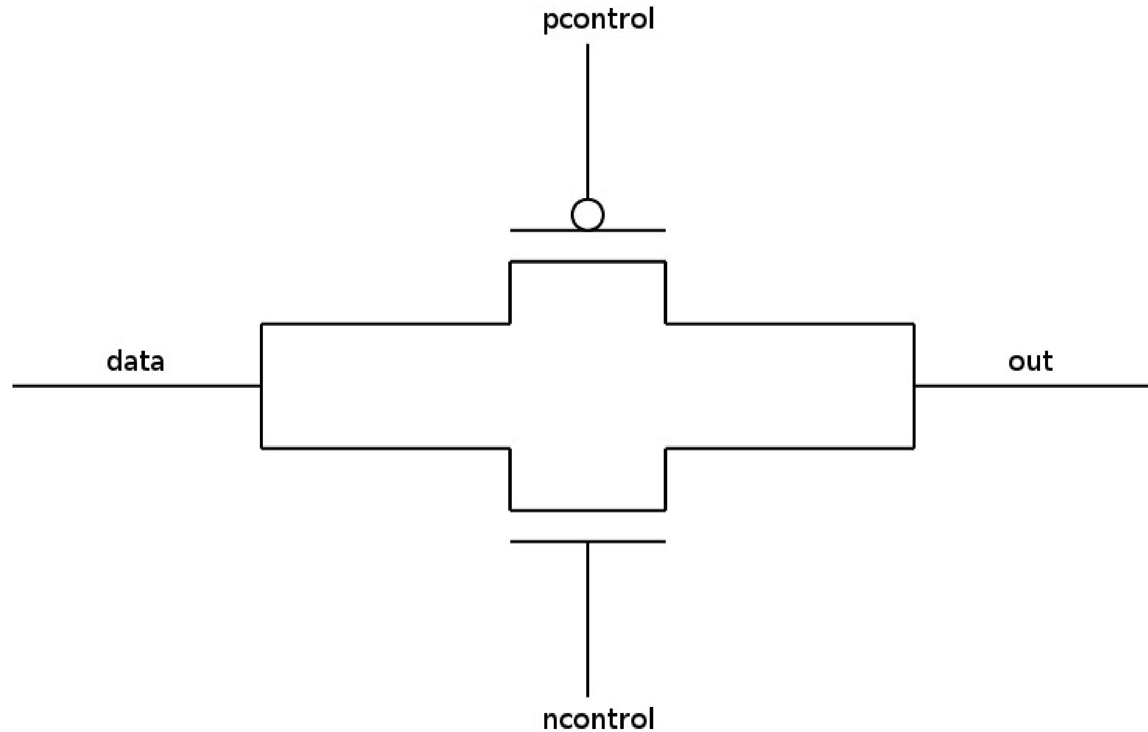


Figura 8.2 Elementul de comutare CMOS

Instantierea unui element de comutare CMOS se face astfel:

```
cmos  c1(out,data,ncontrol,pcontrol);//instantierea  unei  porti
                                     //CMOS
cmos(out,data,ncontrol,pcontrol); //instantierea  unei  porti
                                  //CMOS fara nume de instanta
```

In mod obisnuit semnalele ncontrol si pcontrol sunt complementare:
- cand ncontrol=1 si pcontrol=0 elementul CMOS conduce
- cand ncontrol=0 si pcontrol=1 elementul CMOS comuta iesirea in inalta impedanta

Se poate observa ca elementul CMOS este alcatuit dintr-un tranzistor nmos si unul pmos, astfel ca un element CMOS este echivalent cu:

```
nmos(out,data,ncontrol);   //instantierea unui tranzistor nmos
pmos(out,data,pcontrol);   //instantierea unui tranzistor pmos
```

=> **Elemente de comutare bidirectionale**

Tranzistorii NMOS, PMOS si elementul CMOS conduc in sensul de la drena la sursa. In multe situatii este nevoie de elemente de circuit care conduc in ambele sensuri, in acest caz oricare din terminale poate fi terminalul de intrare.

Se definesc in Verilog trei elemente de comutare bidirectionale:

- *tran*
- *tranif0*
- *tranif1*

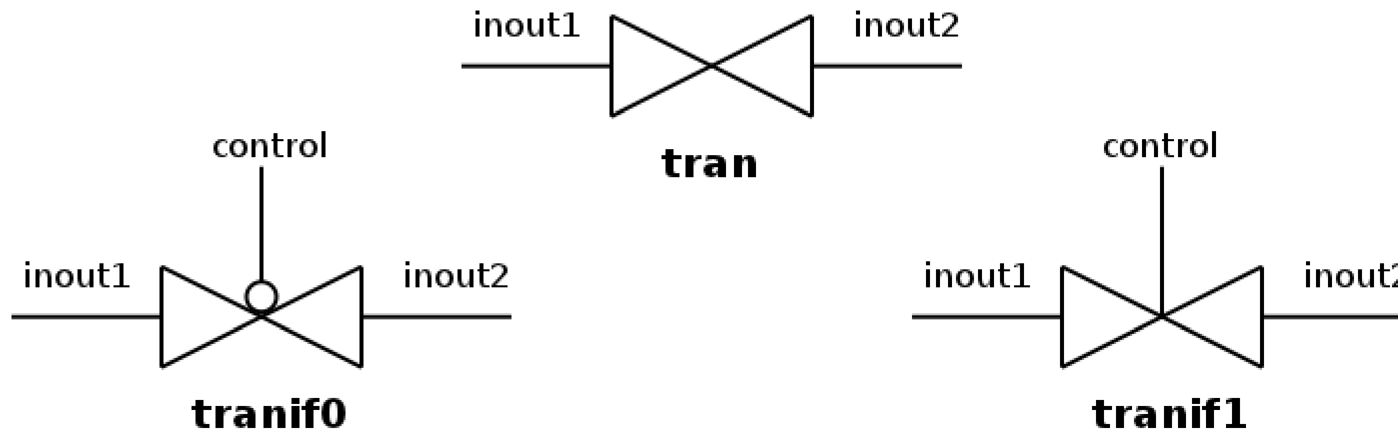


Fig. 8.3 Elemente de comutare bidirectionale

Elementul *tran* actioneaza ca un element de legatura intre doua semnale. Oricare din semnalele *inout1* sau *inout2* poate fi intrare sau iesire. Elementul *tranif0* conecteaza cele doua semnale *inout1* si *inout2* numai atunci cand semnalul de comanda (*control*) este zero. In mod complementar, elementul *tranif1* conduce daca semnalul de comanda este 1. Nivelurile logice asociate acestor elemente se pot deduce din cele asociate tranzistorilor nmos si pmos.

Exemplu:

```
tran t1(inout1,inout2);//numele instantei, 't1' este optional
tranif0(inout1,inout2,control);   //nu se specifica numele
                                  //instantei
tranif1(inout1,inout2,control);   //nu se specifica numele
                                  //instantei
```

Elementele de comutare bidirectionale se folosesc uzual la izolarea intre magistrale sau semnale.

=> **Surse de energie**

La modelarea cu ajutorul tranzistorilor este necesara prezenta unei surse de energie (Vdd, asociata cu un nivel logic 1) si a nivelului de referinta (GND, nivel logic 0). Cuvintele cheie utilizate sunt:

- Vdd => *supply1*
- GND => *supply0*

Exemplu:

```
supply1 vdd;
supply0 gnd;
assign a=vdd; //conecteaza a la vdd
assign b=gnd;//conecteaza b la gnd
```

=> **Intarzieri asociate elementelor de comutare**

Intarzierile care pot fi asignate tranzistorilor sunt asemanatoare cu cele pentru portile logice, avand urmatoarea sintaxa:

Element de comutare	Intarzieri posibile	Exemplu
pmos, nmos	zero(fara intarziere)	pmos p1(out,data,control);
	o intarziere	pmos #(1) p1(out,data,control);
	doua intarzieri	nmos #(1,2) p2(out,data,control);
	trei intarzieri	nmos #(1,3,2) p2(out,data,control);
cmos	Idem	cmos #(5) c2(out,data,nctrl,pctrl);
		cmos #() c2(out,data,nctrl,pctrl);
tran	NU se asociaza nicio intarziere	
tranif1, tranif0	zero, una sau doua intarzieri	tranif0 t1(inout1,inout2,ctrl);
		tranif0 #(3) t1(inout1,inout2,ctrl);
		tranif1 #(1,2) t2(inout1,inout2,ctrl);

Exemple de circuite implementate cu ajutorul elementelor de comutare:

Ex.1 Poarta de tip NOR implementata cu ajutorul unui element CMOS

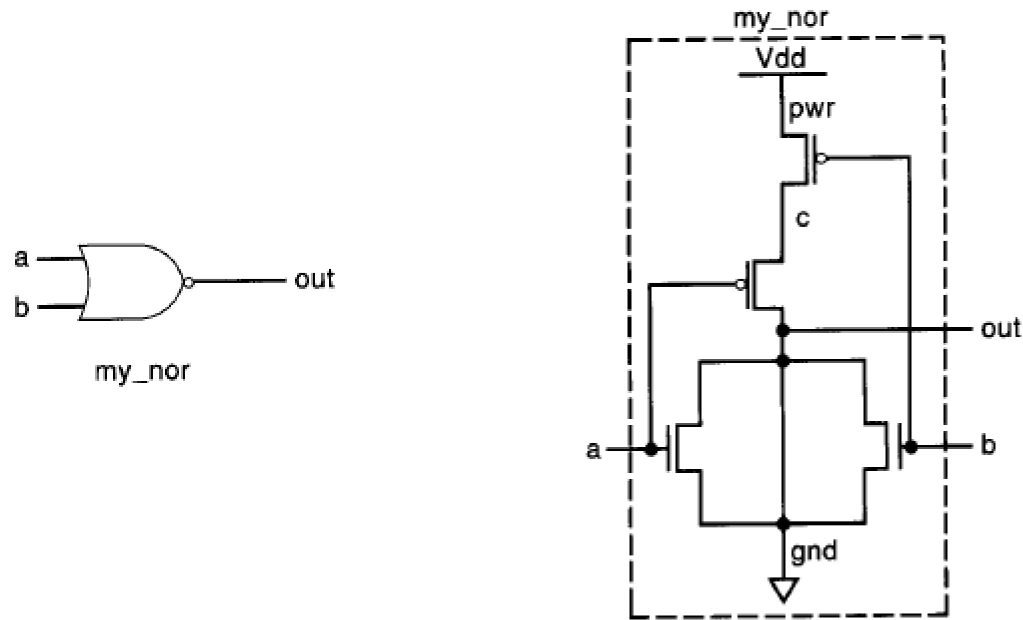


Figura 8.3 Poarta logica si schema cu elemente de comutare

```
//definirea modului my_nor
module my_nor(out,a,b);
    output out;
    input a,b;
    //conectari interne
    wire c;

    //asignarea liniilor de energie si referinta
    supply1 pwr;
    supply0 gnd;

    //instantierea tranzistorilor de comutare pmos
    pmos(c,pwr,b);
    pmos(out,c,a);

    //instantierea tranzistorilor de comutare nmos
    nmos(out,gnd,a);
    nmos(out,gnd,b);
endmodule
```

```
//modulul de test pentru poarta NOR
   module stimulus;
   reg A,B;
   wire OUT;

   //instantiere modul my_nor
   my_nor n1(OUT,A,B);

   //testarea tuturor cazurilor
   initial
      begin
         A=1'b0;B=1'b0;
         #5 A=1'b0;B=1'b1;
         #5 A=1'b1;B=1'b0;
         #5 A=1'b1;B=1'b1;
      end

   initial
      $monitor($time,” OUT=%b A=%b B=%b”,OUT,A,B);
   endmodule
```

```
//rezultatul simularii
   0   OUT=1, A=0, B=0
   5   OUT=0, A=0, B=1
   10  OUT=0, A=1, B=0
   15  OUT=0, A=1, B=1
```

Ex.2 Multiplexor 2:1 definit cu elemente CMOS

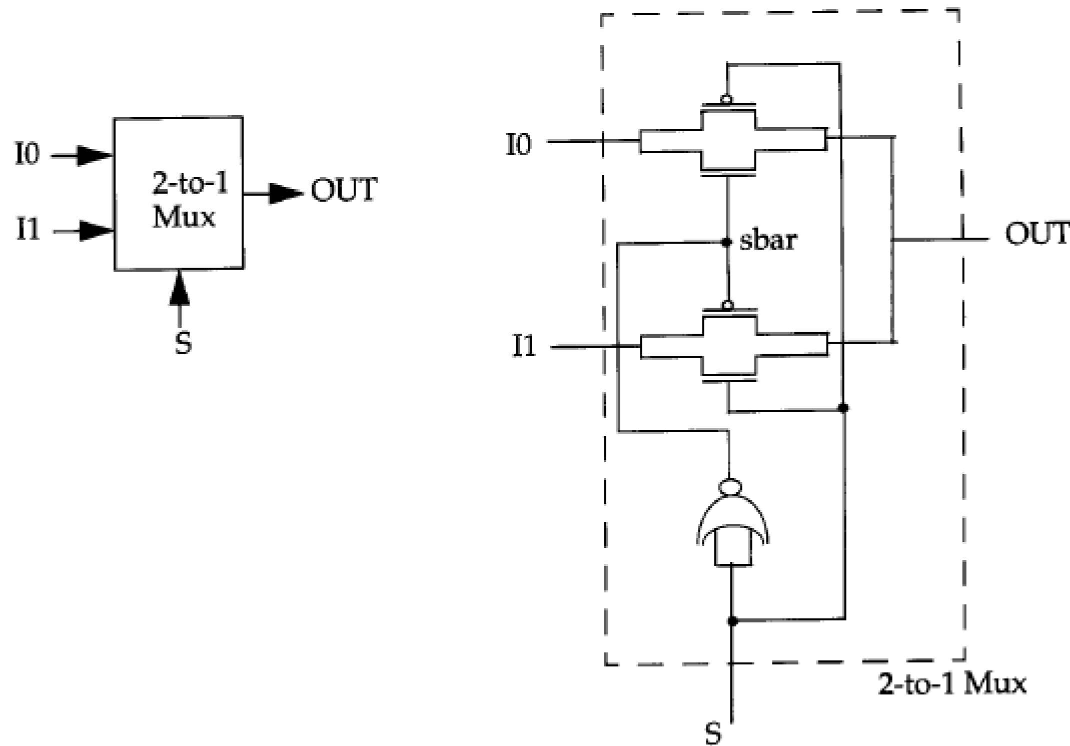


Figura 8.4 Multiplexor 2:1 cu elemente de comutare CMOS

Multiplexorul prezentat transmite la iesirea OUT intrarea I0 daca intrarea de selectie S este 0, altfel, daca S este 1 transmite la iesirea OUT intrarea I1.

```
//definirea modulului multiplexor 2:1
module my_mux(out,s,i0,i1);
    output out;
    input s,i0,i1;
    //conectare interna
    wire sbar;

    //complementul semnalului s se determina cu poarta my_nor
    //anterior definita

    my_nor nt(sbar,s,s);

    //instantierea elementelor de comutare cmos
    cmos(out,i0,sbar,s);
    cmos(out,i1,s,sbar);
Endmodule
```

! Modulul de simulare este lasat ca exercitiu.

Ex3. Implementarea unui bistabil master-slave de tip D cu elemente CMOS.

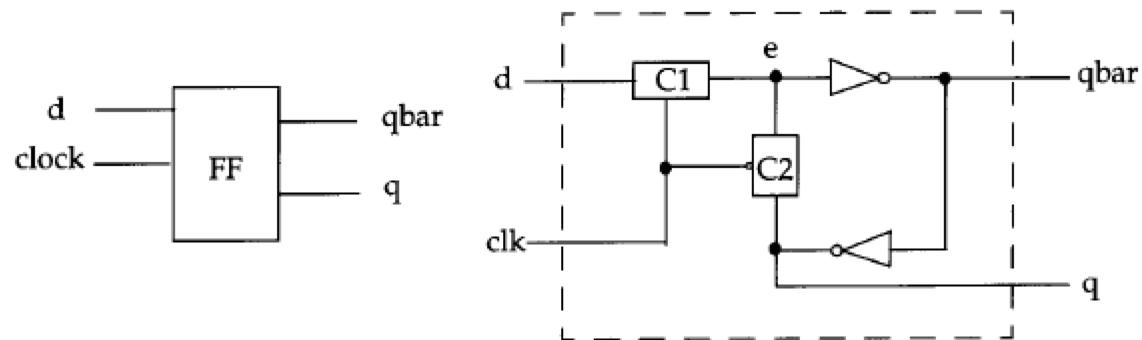


Figura 8.5 Bistabil master-slave CMOS

Elementele C1 si C2 sunt elemente de comutare CMOS. Elementul C1 este deschis daca semnalul *clk*=1, iar C2 este deschis daca semnalul *clk*=0. La intrarea de comanda a elementului C2 este conectat semnalul *clk* complementat.

Construirea inversoarelor CMOS este prezentata in continuare.

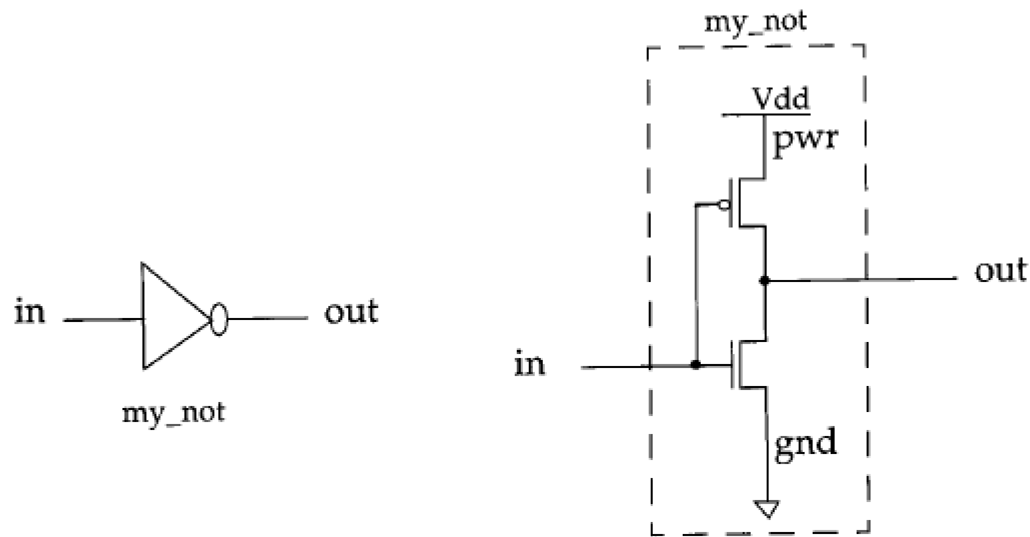


Figura 8.6 Inversorul CMOS

```
//definirea modulului inversor
module my_not(out,in);
    output out;
    input in;

    //declararea surse de energie si a nivelului de referinta
    supply1 pwr;
    supply0 gnd;

    //instantierea tranzistorilor nmos si pmos
    nmos(out,gnd,in);
    pmos(out,pwr,in);
endmodule
```

```
//definirea modulului bistabil D CMOS
module dff(q,qbar,d,clk);
    output q,qbar;
    input d,clk;

    //conectari interne
    wire e;
    wire nclk;

    //instantierea inversorului
    my_not nt(nclk,clk);

    //instantierea portilor CMOS
    cmos(e,d,clk,nclk); //C1 se inchide (e=d) daca clk=1
    cmos(e,q,nclk,clk); //C2 se inchide (e=q) daca clk=0

    //instantiera inversoarelor
        my_not nt1(qbar,e);
        my_not nt1(q,qbar);
endmodule
```

! Modulul de simulare este lasat ca exercitiu.