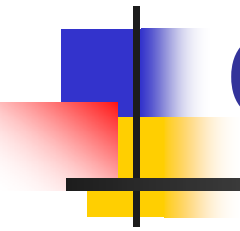
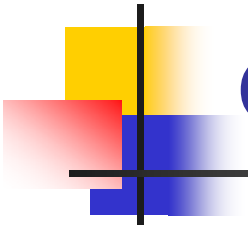


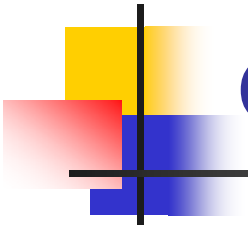
# Circuite integrate digitale

Curs 4



# Curs 4

- circuite secvenţiale
- circuite de memorare simple
  - latch-uri
  - bistabile
  - registre
- exemple de circuite secvenţiale
  - registrul de deplasare
  - numărătorul
- **Aplicaţii**
- Descrierea latch-urilor in Verilog HDL

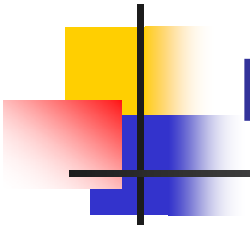


# Circuite secvenţiale

- diferenţa combinaţional – secvenţial
- funcţia de memorare
  - bucla de reacţie
- semnalul de ceas – control în timp (discret)

# Clasificare formală a sistemelor digitale

- Sisteme de ordin... SO
  - SO0 – combinaţionale
  - SO1 – memorii
  - SO2 – automate
  - SO3 – procesoare
- trecerea la un ordin superior: introducerea unei bucle de reacţie



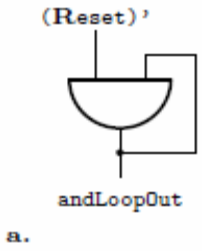
# Funcţia de memorare

- cum putem folosi circuitele logice pentru a memora?
- bucla de reacţie introduce o nouă funcţionalitate

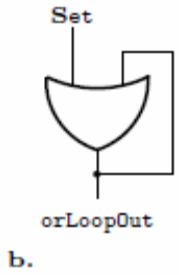
- Denumiri:
  - latch
  - bistabil (flip-flop, FF)

# Latch-uri elementare

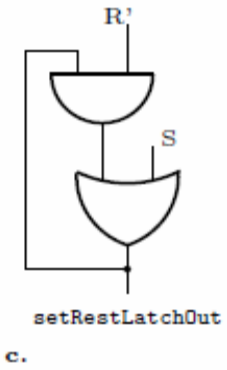
*reset-only* latch

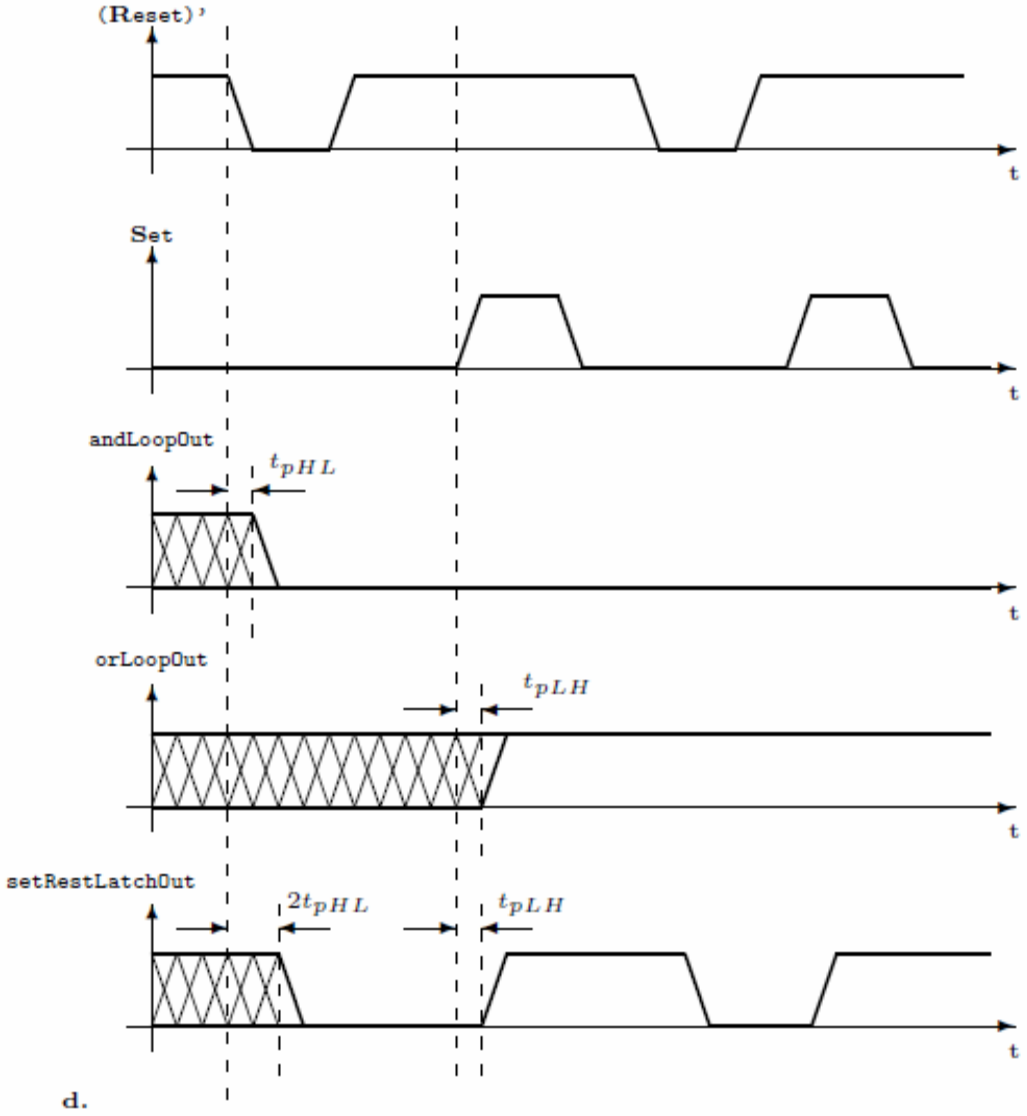



*set-only* latch

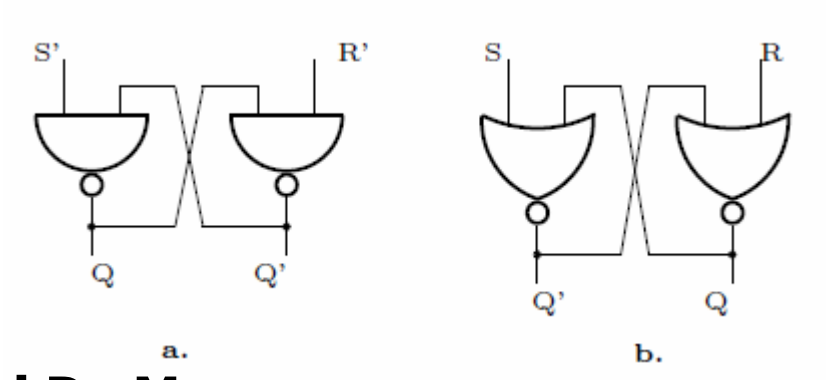



*set-reset* latch



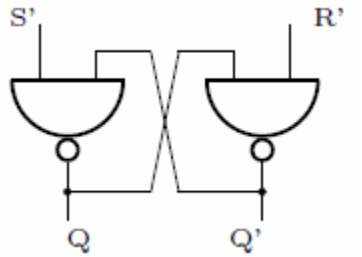



# Latch-uri simetrice




- **Teoremele lui De Morgan**:
  - a+b = (a'b')'
  - ab = (a'+b')'
- Problemele latch-urilor elementare:
  - confuzia comenzi – sincronizare – date
  - combinaţiile "interzise" pe intrare (comportament impredictibil)

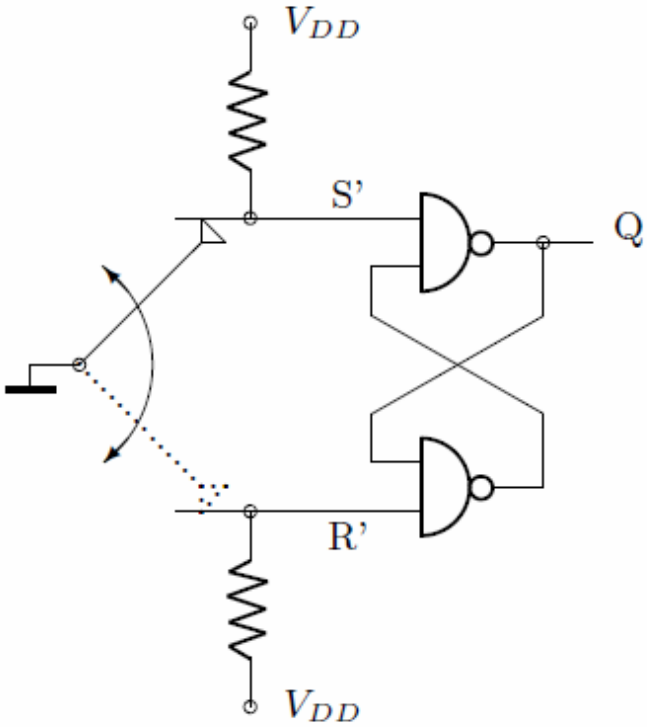
# Tabel de tranziţii

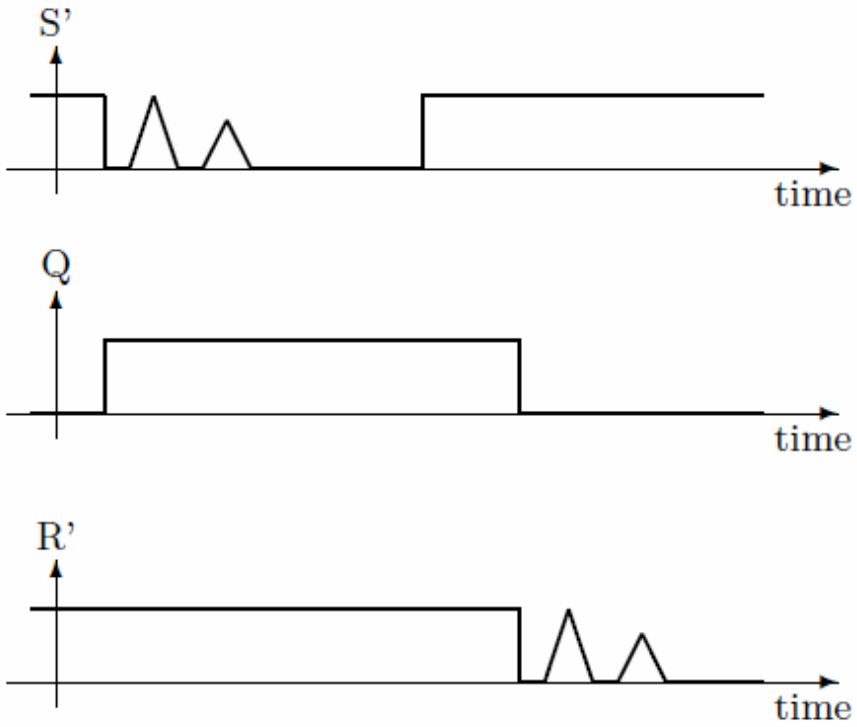



| S' R' | Q+ Q'+ | funcţie |
|---|---|---|
| 1 1 | Q Q' | memorare |
| 0 1 | 1 0 | set |
| 1 0 | 0 1 | reset |
| 0 0 | 0/1 0/1 | ??? |

- memorarea efectivă se face prin secvenţa
  - comanda set sau reset
  - revenire la combinaţia 11
- de exemplu, pentru memorarea valorii 0: 01, 11
- comanda 00 este contradictorie (practic, interzisă)

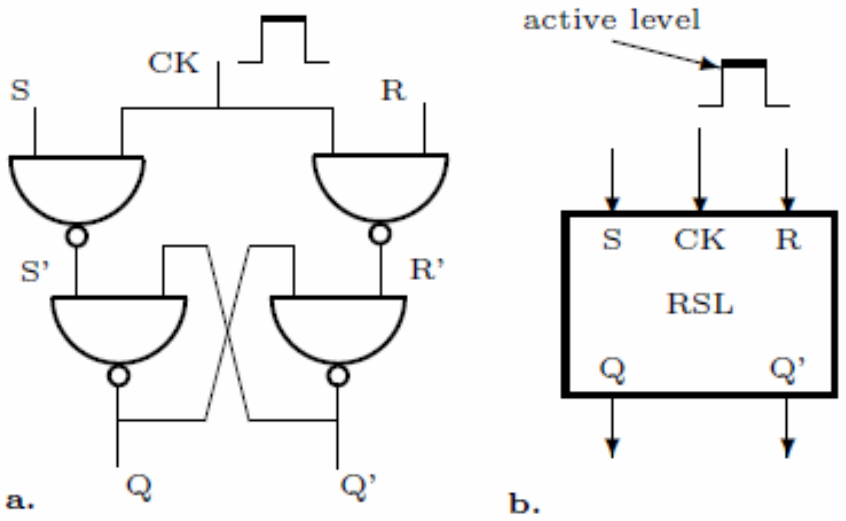
# Aplicaţie: circuitul de debounce



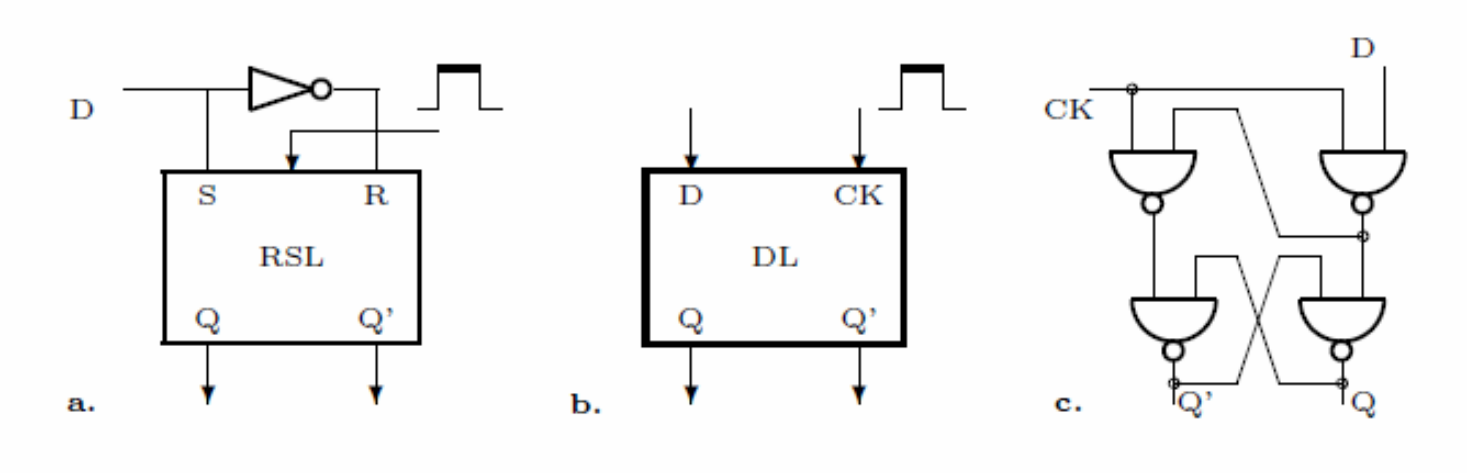
# Latch elementar cu ceas

- semnalul de ceas poate "dezactiva" comenzile (pentru CK=0)
- latch-ul este **transparent** pe palierul de 1 (este identic cu circuitul anterior)



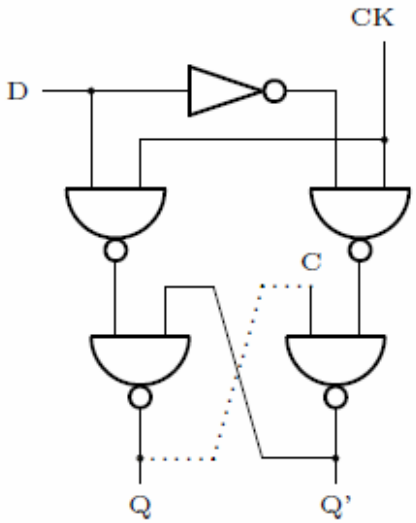
# Latch de date (data latch)

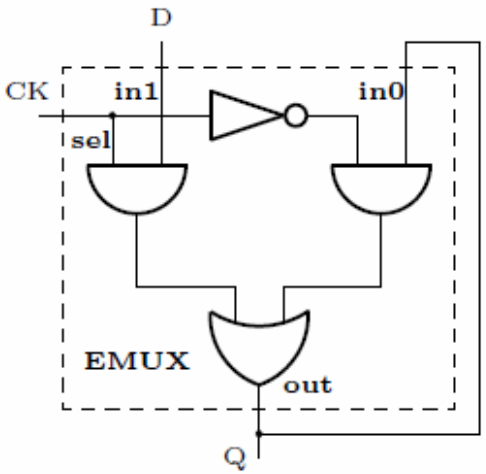
- Dacă R=S’ se elimină comanda contradictorie
- ...dar se elimină şi comanda 00!
- latchul este transparent pe palierul activ al ceasului, ieşirea urmează intrarea
- poate fi controlat semnalul de ceas

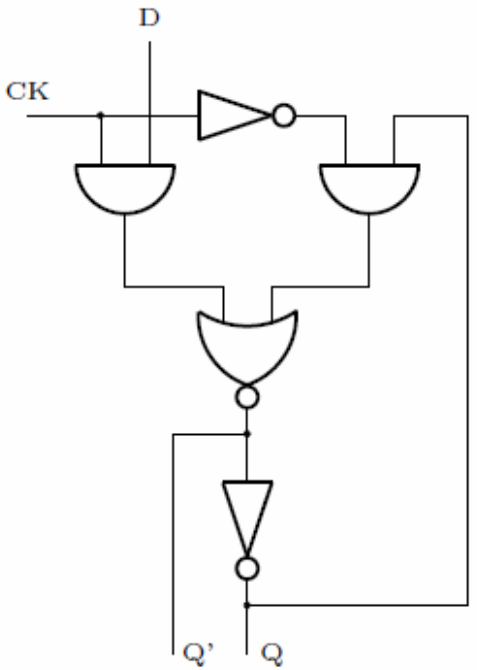


# Tema 4 - variante data latch

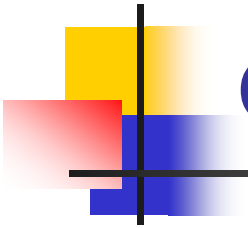
- Scrieţi ecuaţiile logice care descriu funcţionarea schemelor următoare şi demonstraţi că latchul D se poate implementa cu fiecare dintre aceste scheme. Care este optimă?



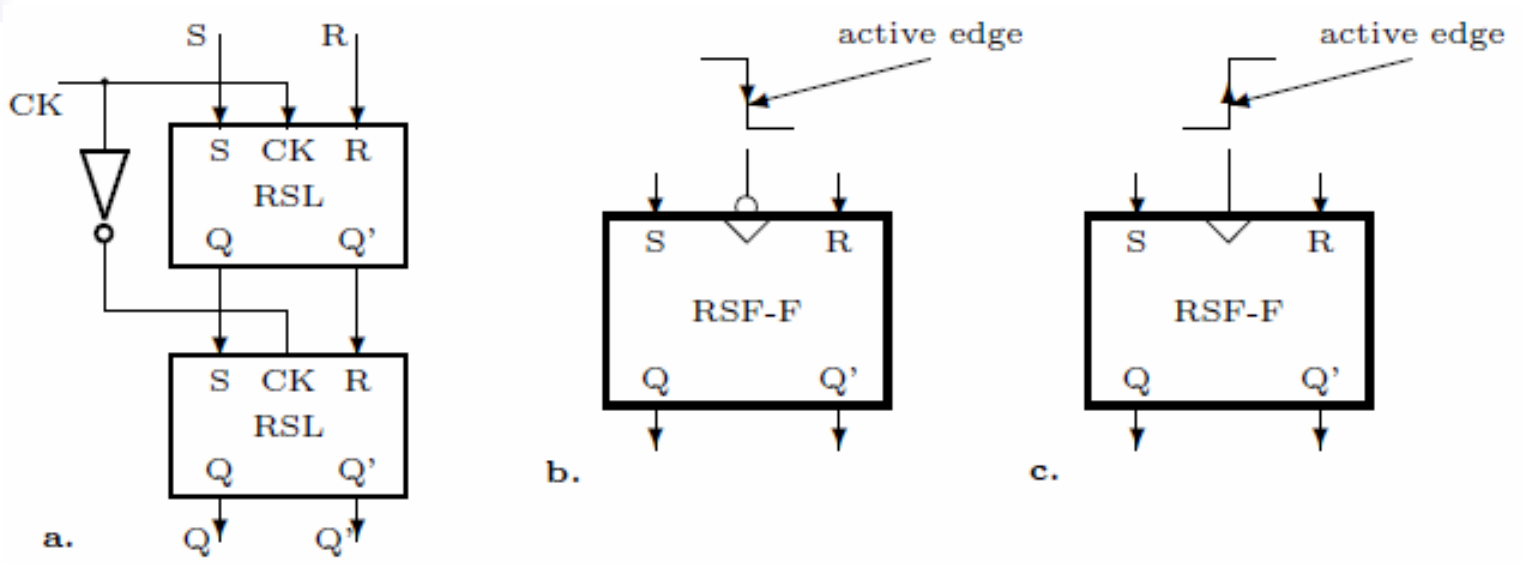



# Concluzii la latch-urile elementare



- Funcţia de memorare se face prin blocarea circuitelor în starea dorită
- Dezavantaj: nu există separarea clară între semnalele care determină **cum** şi **când** comută circuitul
  - latch-ul cu ceas rezolvă parţial problema
- Dezavantaj: comanda interzisă
  - latch-ul D **evită** problema (nu o rezolvă)

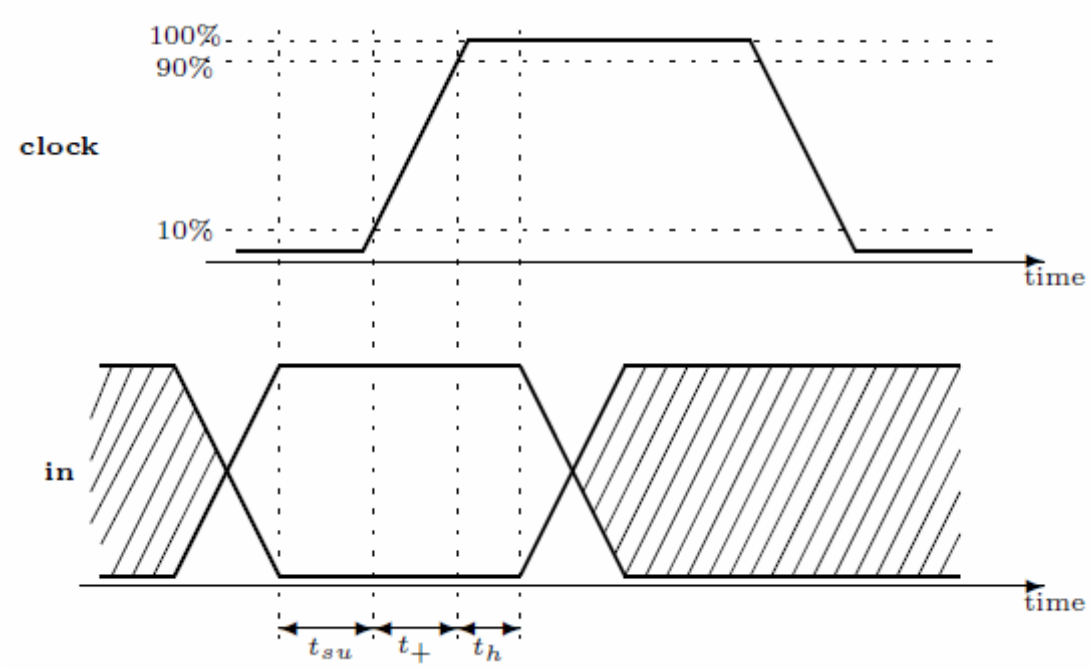
# Principiul master-slave




- două latch-uri cu ceasul în antifază
  - CK = 1 este transparent latch-ul master
  - CK = 0 este transparent latch-ul slave
- bistabilul care rezultă nu este transparent! (dacă datele sunt stabile în momentul tranziţiei ceasului)

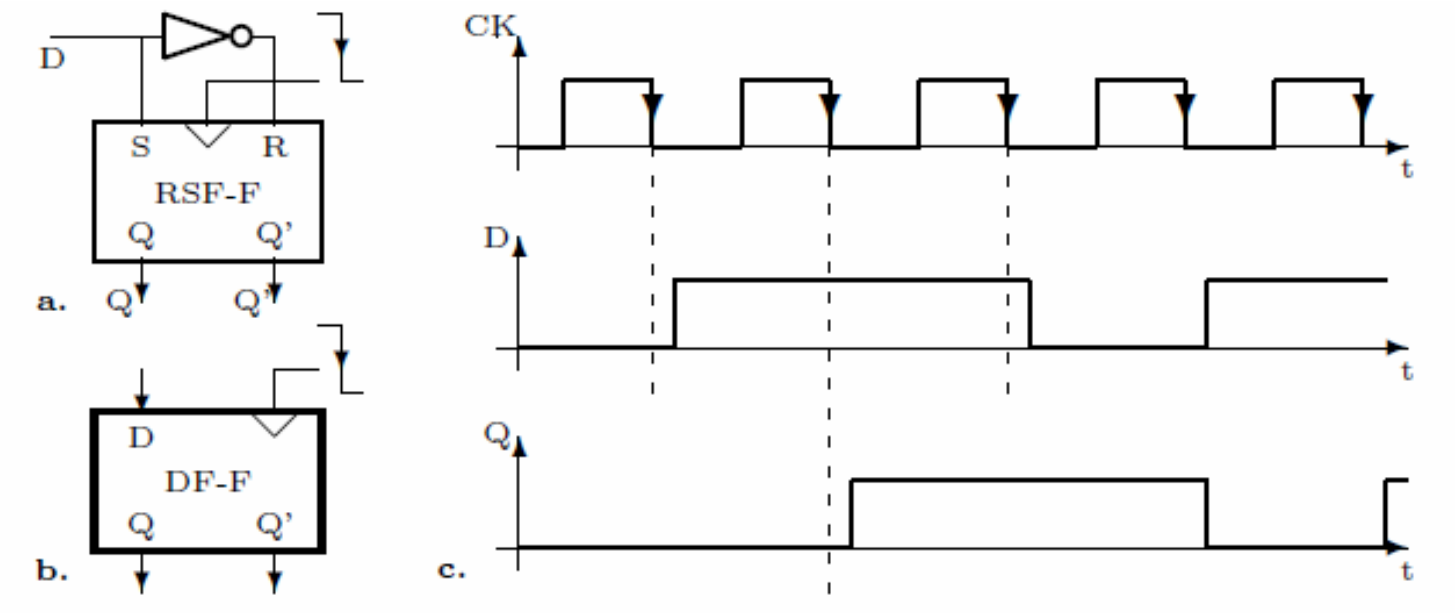
# Ce înseamnă "moment"?

- "moment" = $t_{su} + t_{+} + t_{h}$



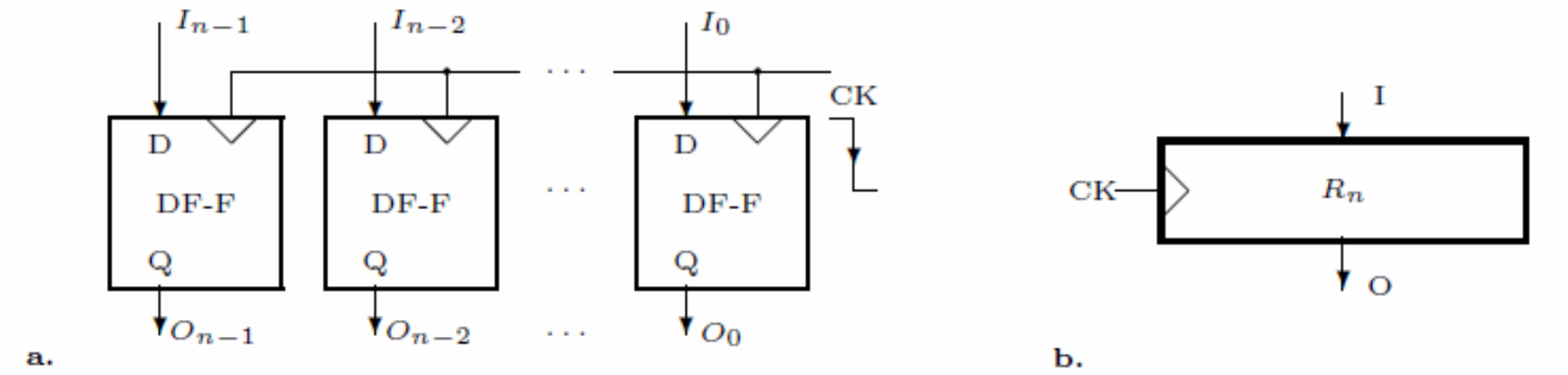
# Bistabilul D (Delay Flip-Flop)

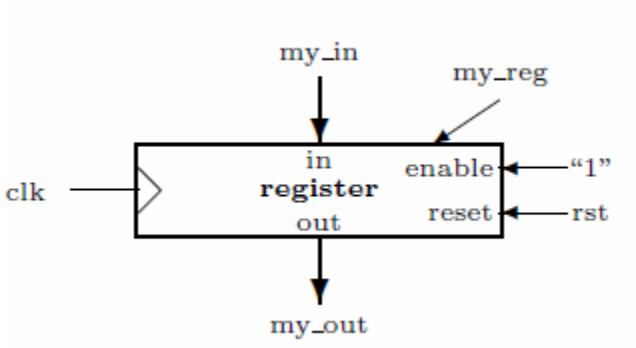
- iesirea urmăreste modificările intrării, sincronizat cu ceasul (este întârziată – **delay**)

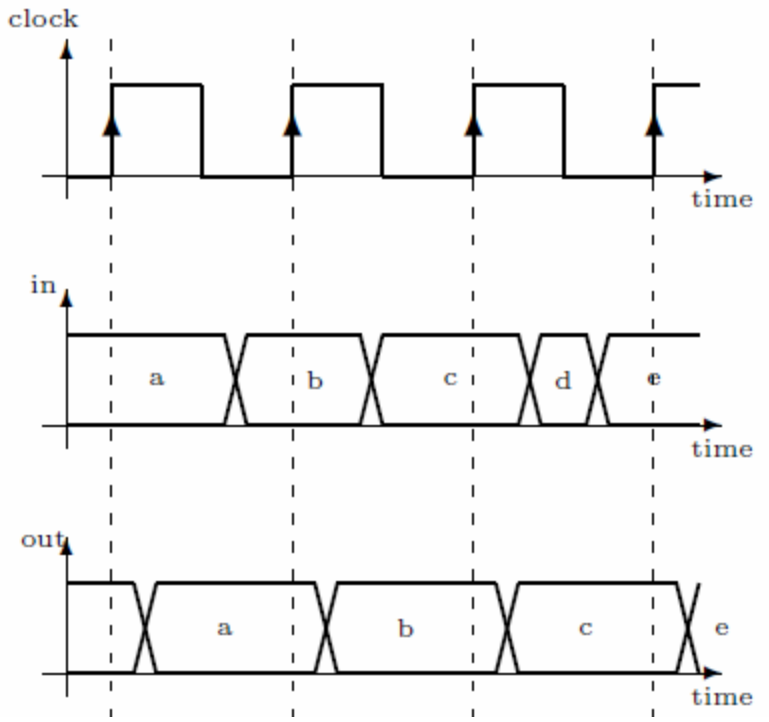


# Registrul

- extindere paralel a bistabilului D
- aplicații: memorare, sincronizare, întârziere, introducerea buclelor de reacție, buffere




my_in
my_reg
clk
in
register
out
enable
reset
"1"
rst
my_out


clock
time
in
a
b
c
d
e
time
out
a
b
c
e
time

# Registrul de deplasare

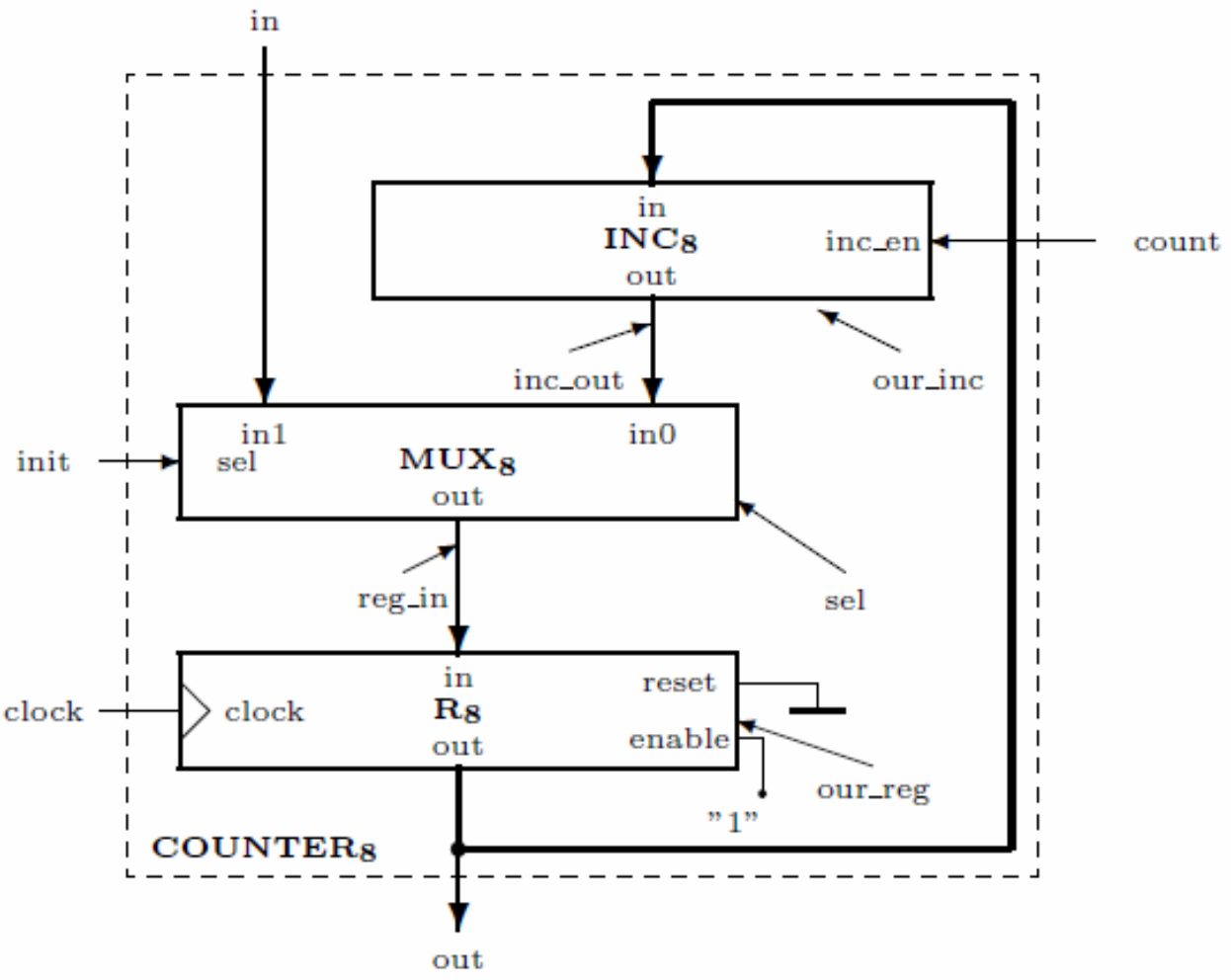
- **00: nop**: no operation, funcţie obligatorie
- **01: load**: initializarea stării registrului
- **10: leftShift**: deplasare la stânga cu o poziţie
- **11: rightShift**: deplasare la dreapta cu o poziţie

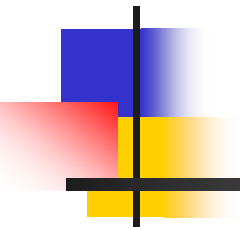
```
module shiftRegister(output reg [15:0] out   ,
                     input      [15:0] in    ,
                     input      [1:0]  mode  ,
                     input             clock);
    always @(posedge clock)
     case(mode)
         2'b00: out <= out;
         2'b01: out <= in ;
         2'b10: out <= out << 1;
         2'b11: out <= out >> 1; // for positive integers
         //2'b11: out <= {out[15], out[15:1]}; // for signed integers
     endcase
endmodule
```

# Numărătorul

```
module counter(    output  reg [7:0]   out     ,
                   input       [7:0]   in      ,
                   input               init    , // initialize with in
                   input               count   , // increment state
                   input               clock   );
   always @(posedge clock)              // always at the positive edge of clock
       if (init)           out <= in;
           else if (count) out <= out + 1;
endmodule
```

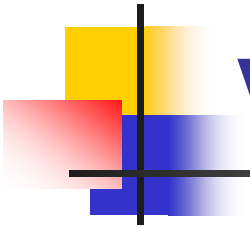
- if...else sugerează un multiplexor
- out = out + 1 incrementare


in
in
INC8
out
inc_en
count
inc_out
our_inc
in1
sel
MUX8
in0
init
out
reg_in
sel
clock
clock
in
R8
out
reset
enable
our_reg
"1"
COUNTER8
out



# Verilog
# Latch-uri şi bistabile

Modele comportamentale



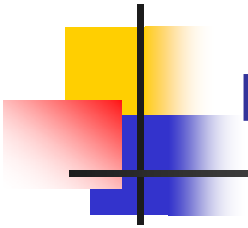
# Variabile de tip reg

- net-urile (cum sunt definite implicit porturile; variabilele de tipul **wire**) nu sunt suficiente!
- de ce?

  pentru că driverul trebuie să fie activ tot timpul

  pentru că nu pot implementa algoritmi

**Variabilele reg memorează valorile între două atriburi succesive**



# reg

**Definiţie:** *O variabilă* reg *este o abstracţie a unui element de stocare a datelor.*

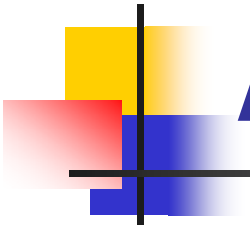
(este analog variabilelor în limbajele de programare C sau Pascal)

Nu este analogul unui registru fizic, deoarece nu există semnal de ceas. Valoarea stocată este memorată până la următoarea atribuire explicită.

Înainte de prima atribuire, are valoarea x (înainte de prima atribuire, neturile au valoarea z).

# Tipuri de variabile reg

| | |
|---|---|
| **reg** | Variabile simple, stochează de obicei valori logice. Implicit, pe un bit. |
| **time** | Registru special care stochează timpul (de simulare) – registru pe 64 de biţi. |
| **integer** | Registru folosit pentru stocarea numerelor (întregi cu semn). Minim 32 de biţi. |
| **real** | Registru folosit pentru stocarea numerelor reale. Set restricţionat de operaţii. |
| **realtime** | Stocarea timpului, definit ca număr real. |



# Atribuiri

- pentru semnale, variabile…

- assign (atribuire continuă), pentru net-uri
- = atribuire folosită pentru reg-uri
- <= *non-blocking* (tot pentru reg-uri)

# Diferenţa dintre = şi <=

| = | <= |
|---|---|
| atribuire procedurală<br>*blocked assignement* | atribuire concurentă<br>*unblocked* |
| instrucţiunea care conţine = trebuie să se încheie înainte de a se trece la următoarea | toate instrucţiunile care conţin <= se execută în paralel |

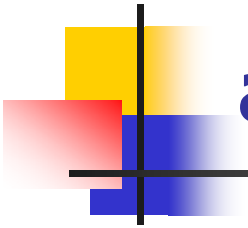
# Registru de deplasare

```
always@(posedge clk or
   posedge rst) begin
if (rst) begin
   A=0;B=0;C=0;D=0;
else begin
   A=B;
   B=C;
   C=D;
   D=E;
end
end
```

CORECT

```
always@(posedge clk or
   posedge rst) begin
if (rst) begin
   A=0;B=0;C=0;D=0;
else begin
   D=E;
   C=D;
   B=C;
   A=B;
end
end
```

GREŞIT!!!



# always @

- sensibil la palier (valori logice): circuite combinaţionale şi latchuri transparente

  always @ (a or b or c)

  SE RECOMANDĂ FOLOSIREA =

- sensibil la front (modificarea valorilor semnalelor): bistabile, circuite secvenţiale sincrone

  always @ (posedge ck)

  SE RECOMANDĂ FOLOSIREA <=

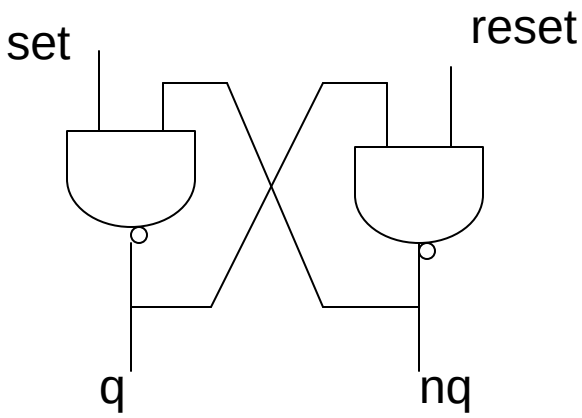
# Latch cu assign

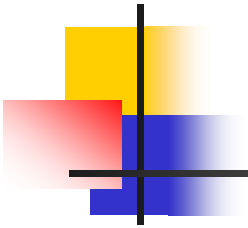
latch cu 2 porţi nand

1. definire structurală
   (funcţionează, dar oscilează)
2. definire data-flow

```
assign q = set ~& nq;
assign nq = reset ~& q;
```

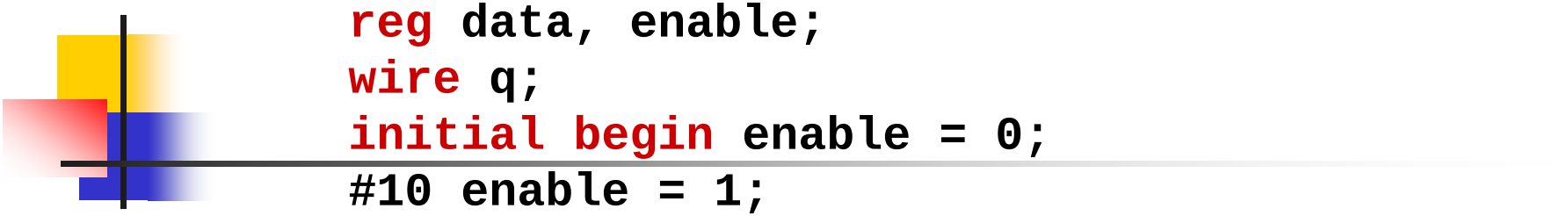
**NU funcţionează!!!**



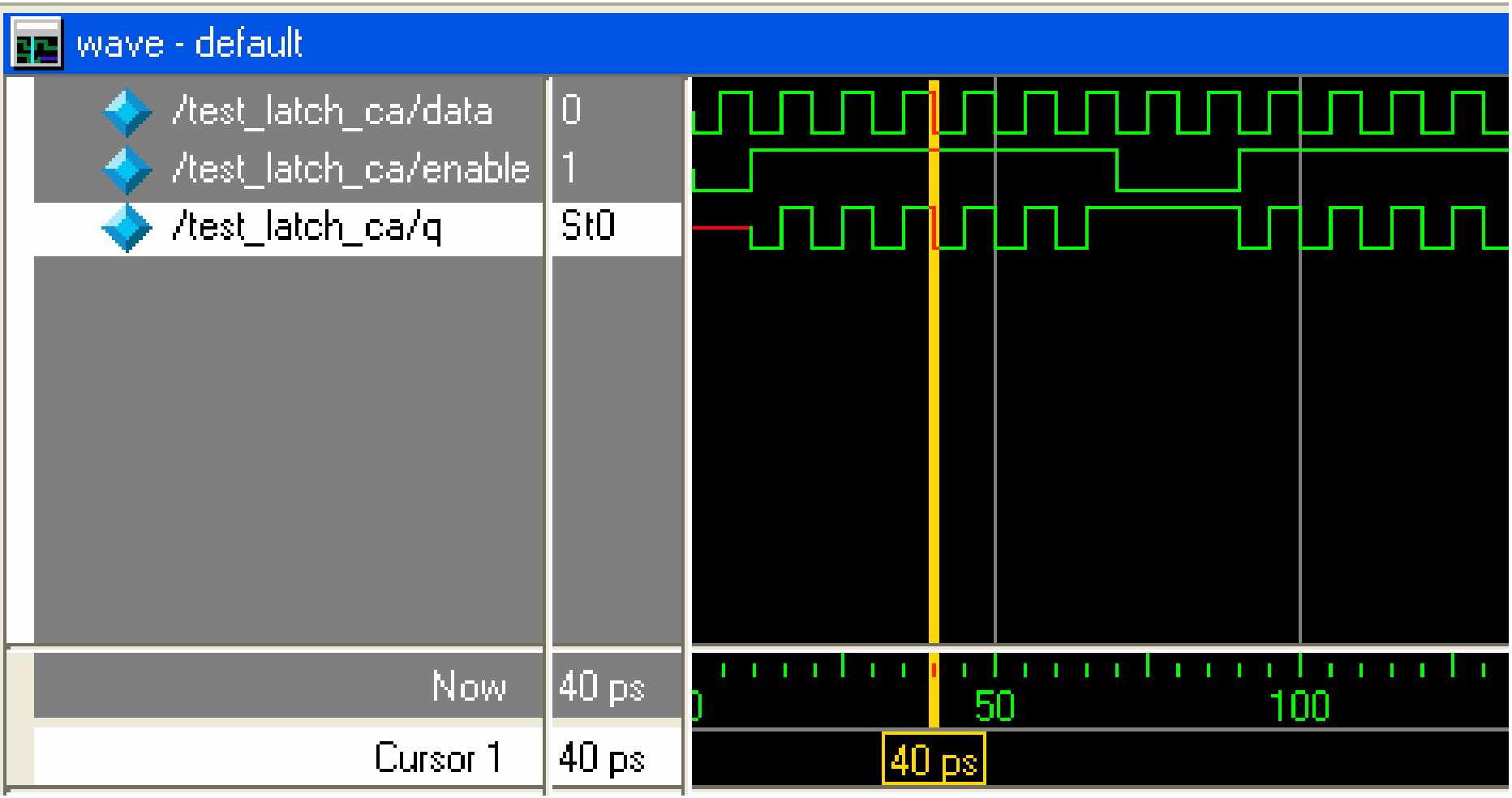


# Latch transparent (cu assign)

```
module latch_CA(q_out, data_in, enable);
    output q_out;
    input data_in, enable;

    assign q_out = enable ? data_in : q_out;
endmodule
```



```
module test_latch_ca;
    reg data, enable;
    wire q;
    initial begin enable = 0;
    #10 enable = 1;
    #60 enable = 0;
    #20 enable = 1;
    #50 $stop;
end

initial begin data = 0;
forever #5 data = ~data;
end

latch_CA my_latch(q, data, enable);

endmodule
```

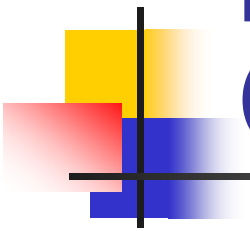

wave - default
/test_latch_ca/data
/test_latch_ca/enable
/test_latch_ca/q
0
1
St0
Now
40 ps
Cursor 1
40 ps
50
100
40 ps

# Alt exemplu

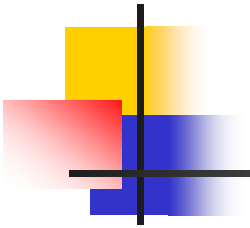
```
module latch_reset(q_out, data_in, enable, reset);
      output q_out;
      input data_in, enable, reset;

      assign q_out = !reset ? 0 : enable? data in :
  q_out;

endmodule
```

**explicaţi funcţionarea acestui latch!**

# Modele comportamentale ciclice (pentru latch-uri şi bistabile)

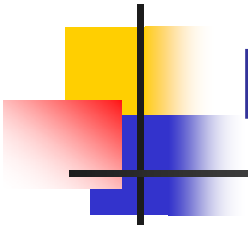


- instrucţiunea assign modelează comportamentul sensibil la nivelurile logice
- NU poate modela fronturile (comportamentul determinat de tranziţiile semnalelor)
- circuitele SINCRONE (definiţie): toate evenimentele se produc ca urmare a frontului activ al ceasului



# Exemplu de latch

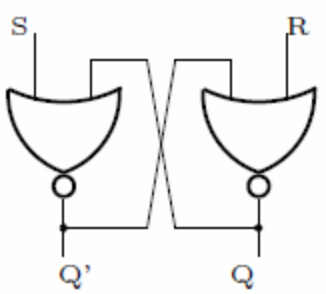
```
always @(posedge <clock> or posedge <reset>)
begin
    if (<reset>) begin
      <reg> <= 1'b0;
    end
    else begin
      <reg> <= <signal>;
    end
```



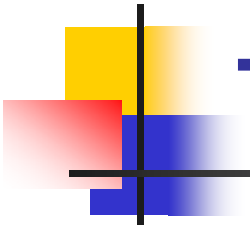
# Exemplu de latch (2)

```
always @(posedge <clock>) begin
    if (<reset>) begin
      <reg> <= 1'b0;
    end
    else if (<clock_enable>) begin
      <reg> <= <signal>;
    end
```

# Aplicaţii




- Completaţi tabelul de tranziţii pentru latch-ul elementar cu porţi NOR
- Desenaţi schema internă a unui registru de deplasare



# Tema 5

- Desenaţi schema internă a unui registru serie-paralel pe 4 biţi care are următoarele funcţii:
  - NOP no operation
  - incarcare paralel
  - deplasare la stânga o poziţie
- Completaţi schema anterioară cu a patra funcţie, deplasare la dreapta cu o poziţie

- **indicaţie**: folosiţi multiplexoare pentru a selecta valoarea încărcată în fiecare bistabil