# Laboratorul 4 - Limbajul Verilog: Circuite secvențiale - Partea I

În laboratoarele anterioare au fost prezentate construcțiile Verilog pentru descrierea comportamentală a circuitelor combinaționale, ilustrate în exemplul următor. Laboratorul curent va prezenta elementele folosite pentru decrierea comportamentală a circuitelor secvențiale:

- blocuri *always@* edge-triggered
- atribuiri non-blocante (<=)
- implementarea automatelor de stări finite

## Blocul always@ edge-triggered

În laboratoarele anterioare blocul `always` a fost prezentat drept o porțiune de cod ce modelează un anumit comportament, și care se execută ciclic la schimbarea valorii unor semnale.

În afară de circuitele care depind doar schimbarea nivelului semnalului, există și circuite al căror comportament depinde de tranzițiile semnalului (activ pe *front crescător* sau *front descrescător*). Starea bistabililor, de exemplu, se modifică pe frontul crescător sau descrescător al unui semnal de ceas. În cazul acesta, blocul `always@` trebuie să se execute la detecția unui astfel de front (eng. *edge-triggered*). Pentru a modela un astfel de comportament Verilog oferă cuvântul cheie **posedge** ce poate fi alăturat numelui semnalului unui semnal din lista de senzitivități pentru a indica activarea blocului `always` la un front al semnalului. De exemplu blocul “always @(posedge clk)” se activează pe frontul crescător al semnalului *clk*.

### Sensitivity list

Cuvintele cheie **posedge** (pentru front crescător) și **negedge** (pentru front descrescător) indică activarea blocului *always@ edge-triggered* la schimbarea frontului semnalului.

### Exemplu sensitivity list

```
always @(posedge sig)   // frontul crescător al semnalului 'sig'
always @(negedge sig)   // frontul descrescător al semnalului 'sig'
always @(posedge sig1, posedge sig2)    // frontul crescator al
// semnalului 'sig1' sau frontul crescător al semnalului 'sig2'
always @(posedge sig1, negedge sig2)    // frontul crescător al
// semnalului 'sig1' sau frontul descrescător al semnalului 'sig2'
```

### Atribuiri non-blocante

În blocurile `always@` din laboratoarele precedente au fost folosite atribuirile ce utilizează operatorul "=", numite *atribuiri blocante*, deoarece se execută secvențial, ca în limbajele de programare procedurale (C, Java etc). Verilog oferă și un alt tip de atribuiri, care sunt executate toate în același timp, în paralel, indiferent de ordinea lor în bloc. Pentru a descrie un astfel de comportament se folosește operatorul "⇐", iar atribuirile se numesc *atribuiri non-blocante*. Acest nou tip de atribuire **modelează concurența care poate fi întâlnită în hardware la transferarea datelor între registre**.

Variabilele cărora li se atribuie o valoare trebuie să fie de tip registru (*reg, integer*) atât în cazul blocant cât și în cel non-blocant. Simulatorul evaluează întâi partea dreaptă a atribuirilor și apoi atribuie valorile către partea stângă. Acest lucru face ca ordinea atribuirilor non-blocante să nu conteze, deoarece rezultatul lor va depinde de ce valori aveau variabilele din partea dreaptă înainte de execuție.

### Exemplu atribuiri non-blocante

```
always @(posedge sig) // executat pe frontul crescător al semnalului sig
begin
    a <= b;
    b <= a;     // se interschimba valoarea lui a cu cea a lui b
    c <= d;     // toate trei atribuirile au loc în același timp
end
```

În cadrul blocurilor always care modelează logică **combinațională** se folosesc **atribuiri blocante** ("="), iar în blocurile care modelează logică **secvențială** se folosesc **atribuiri non-blocante** ("⇐")

## Bistabilul D

Exemplele următoare reprezintă implementarea unui bistabil D, prezentat în laboratorul 0, care menține valoarea de intrare ("D") între două fronturi crescătoare ale semnalului de ceas ("clk"). Circuitului prezentat în laboratorul 0 i s-a adăugat și un semnal de reset ("rst_n"). Numele semnalului de reset se termină cu "_n", în mod convențional, pentru a sugera că acesta este activ pe negedge.

În exemplul de mai jos, semnalul de reset este verificat **sincron**, atribuirile făcute ieșirii Q fiind **non-blocante**. Observați că operația de reset este condiționată de valoarea "0" a semnalului "rst_n".

## Bistabilul D - reset verificat sincron

```
module D_flip_flop(output reg Q, input D, clk, rst_n);

always @(posedge clk) begin
    if(!rst_n)
      Q <= 0;
    else
      Q <= D;
end

endmodule
```

Verificarea resetului se poate realiza și în mod asincron.

## Informații adiționale despre always asincron

În cel de-al doilea exemplu, semnalul este verificat **asincron**. Modulul este sintetizabil și are un comportament asemănător cu modulul asincron din al treilea exemplu. Pentru a fi sintetizabil este necesar ca toate atribuirile asupra registrului Q să fie realizate în același bloc *always*, iar blocul *always* să fie activat pe *frontul crescător* al semnalului *clk* sau pe *frontul crescător* al semnalului *!rst_n*.

## Bistabilul D - reset verificat asincron

```
module D_flip_flop(output reg Q, input D, clk, rst_n);

always @(posedge clk or negedge rst_n) begin
    if(!rst_n)
      Q <= 0;
    else
      Q <= D;
end

endmodule
```

Un **modul** este **nesintetizabil** dacă acesta conține atribuiri asupra aceluiași registru în mai mult de un bloc **always**.

În cel de-al treilea exemplu, este prezentat cazul în care semnalul de reset este verificat **asincron**, iar atribuirile făcute ieșirii Q sunt **blocante** în cazul în care semnalul ”!rst_n” devine 1 logic sau **non-blocante** pe frontul crescător al semnalului “clk”. În acest caz, se obține un modul nesintetizabil.

## Bistabilul D - reset verificat sincron (modul nesintetizabil)

```
always @(posedge clk) begin
    if(rst_n)
       Q <= D;
end

always @(*) begin
    if(!rst_n)
       Q <= 0;
end

endmodule
```

## Automate finite

Automatele finite (eng. Finite State machine - FSM), amintite în laboratorul 0, sunt implementate prin logică secvențială. Știind comportamentul unui anumit automat, îl putem implementa folosind două blocuri always@ care să modeleze partea de stare și, respectiv, logica combinațională a acestuia.

Elementele de memorare (stare) ale circuitului se modelează printr-un bloc activ pe frontului semnalului de ceas. În blocul combinațional trebuie tratate toate stările posibile ale automatului, semnalele de ieșire și tranzițiile din aceste stări.

```
module fsm(output reg out, input in, clk, reset_n);
reg [2:0] state, next_state;

// partea secvențială
always @(posedge clk) begin
    if (reset_n == 0) state <= 0;
    else state <= next_state;
end

// partea combinationala
always @(*) begin
    out = 0;
    case (state)
        0:   if (in == 0) begin
                 next_state = 1;
                 out = 1;
             end
             else next_state = 2;
        1:   if (in == 0) begin
                 next_state = 3;
                 out = 1;
```

```
            end
            else next_state = 4;
    ...
    endcase
end
endmodule
```

Nu combinați blocurile secvențiale cu cele combinaționale (e.g. “always @(posedge clk, state, in)”) deoarece majoritatea utilitarelor nu vor sintetiza corect un astfel de circuit.

### Expresii regulate

Expresiile Regulate sunt secvențe de caractere ce definesc un tipar de căutare, folosite în multe cazuri pentru identificarea șirurilor sau sub-șirurilor de caractere ce se potrivesc cu expresia. Cea mai simplă metodă de vizualizare a unei expresii regulate este prin intermediul Automatelor Finite de stări.

Pentru a descrie un tipar care conține un sub-șir între zero și nelimitate ori, este utilizat cuantificatorul “*”, iar pentru a descrie un tipar care conține un sub-șir între una și nelimitate ori, este utilizat cuantificatorul ”+”.

Parantezele ”(” ”)” sunt folosite pentru a delimita grupuri de caractere. Dacă acestea nu sunt specificate, cuantificatorul va avea efect asupra caracterului anterior.

### Exemple:

```
a(bc)*  -  se va potrivi cu șirurile de caractere 'a', 'abc', 'abcbc',
'abcbcbc' etc.
(ab)+c  -  se va potrivi cu șirurile de caractere 'abc', 'ababc', 'abababc'
etc.
ab+a    -  se va potrivi cu șirurile de caractere 'aba', 'abba', 'abbba' etc.
```

Nu confundați operatorii “*” și ”+” cu înmulțire și adunare. În contextul expresiilor regulate, aceștia sunt folosiți pentru a descrie tipare de căutare și **NU** sunt folosiți pentru a scrie cod Verilog.

## Debouncing

Atunci când un buton este apăsat sau un switch este comutat, două părți metalice intră în contact pentru a permite curentului să treacă. Cu toate acestea, ele nu se conectează instantaneu, ci se conectează și deconectează de câteva ori înainte de realizarea conexiunii propriu-zise. Același lucru se întâmplă și în momentul eliberării unui buton (când acesta nu mai este apăsat). Acest fenomen poate conduce la comutări false sau modificări multiple nedorite asupra semnalului și este denumit **bouncing**.

Prin urmare, se poate spune că fenomenul de “bouncing” nu este un comportament ideal pentru niciun switch care execută mai multe tranziții ale unei singure intrări. Aceasta nu este o problemă majoră când avem de-a face cu circuite de putere, dar poate cauza probleme atunci când avem de-a face cu circuitele logice sau digitale. Așadar, pentru a elimina oscilațiile din semnal cauzate de acest fenomen se folosește principiul de **Switch Debouncing**.

Procedeul de debouncing este întâlnit, de asemenea și în software. Urmăriți în scheletul de cod din exercițiul 1 cum este implementat un debouncer și analizați comportamentul acestuia.

# Exerciții

1. Se dorește proiectarea unui automat finit capabil să recunoască secvențe de tip “ba”. Automatul primește la intrare în mod continuu caractere codificate printr-un semnal de un bit (caracterele posibile sunt “a” și “b”). Ieșirea automatului va consta dintr-un semnal care va fi activat (valoarea 1) atunci când la intrare am avut prezent un șir care se potrivește cu tiparul de căutare.
   1. Implementați automatul în Verilog.
      - *Hint*: Realizați pe hârtie schema automatului de stări, pentru a o folosi ulterior ca referință.
      - *Hint*: Observați în laboratorul 0 strategia abordată pentru implementarea unui automat ce recunoaște o secvență de caractere.
   2. Simulați automatul folosind modulul de test din scheletul de cod. Eliminați semnalele nerelevante (*is* și *count*) din diagrama de semnale. Adăugați starea automatului și starea următoare a automatului la diagrama de semnale.
      - *Hint*: Semnalele pot fi eliminate din diagrama de semnale cu *click dreapta→Delete* pe semnalul care se dorește a fi eliminat.
      - *Hint*: Semnale noi pot fi adăugate la diagrama de semnale prin *drag-and-drop* din fereastra *Simulation Objects for ...*, care conține toate semnalele modulului selectat în fereastra *Instance and Process Name*.
      - *Hint*: Simularea trebuie repornită prin *Simulation→Restart* urmat de *Simulation→Run* pentru a vedea comportamentul semnalelor adăugate.
   3. Urmăriți diagrama de semnale și codul automatului și explicați comportamentul. Urmăriți și explicați funcționarea modulului de test.
2. Se dorește realizarea unei treceri de pietoni semaforizate. Duratele de timp pentru cele 2 culori vor fi: roșu - 60 sec, verde - 30 sec.
   1. Implementați și simulați în Verilog automatul necesar. Ce rol are modulul *trecere* din fișierul *trecere.v*?
      - *Hint*: Consultați laboratorul 0 pentru diagrama de tranziție a unui automat similar și propuneți o diagramă de tranziție pretabilă cerinței noastre
   2. Explicați codul numărătorului din fișierul *counter.v*.
      - *Hint*: Urmăriți comportarea acestuia pe diagrama de semnale.

# Resurse

- Schelet de cod
- PDF laborator
- Soluție laborator