

Laboratorul 4 - Limbajul Verilog: Circuite secvențiale

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 descrierea comportamentală a circuitelor secvențiale:

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

Exemplu combinare instanțieri, atribuiri continue, blocuri comportamentale

```
module simple_alu(  
    output reg[7:0] out,  
    input[3:0] a, b,  
    input op);    // operație: 0 - adunare, 1 - înmulțire  
  
    wire[4:0] sum;    // ieșirea sumatorului  
    wire[7:0] sum_ext;    // ieșirea sumatorului extinsă la dimensiunea  
    ieșirii UAL-ului  
  
    wire[7:0] prod;    // ieșirea multiplicatorului  
  
    // sumator 4 biți, ieșire 5 biți  
    adder4 add(sum[3:0], sum[4], a, b, 1'b0);  
  
    // multiplicator 4 biți, ieșire 8 biți  
    multiplier mul(prod, a, b);  
  
    // extinde ieșirea sumatorului, completând cu 0  
    assign sum_ext = {3'b0, sum};  
  
    // selectează ieșirea UAL-ului în funcție de operație  
    always @(*) begin  
        case(op)  
            0: out = sum_ext;  
            1: out = prod;  
            default: out = 8'bx;  
        endcase  
    end  
  
endmodule
```

În exemplul de mai sus au fost folosite atât instanțieri de module, ce țin de o descriere structurală cât

și atribuirii continue și cod procedural, care realizează o descriere comportamentală. Instanțierile de module sau primitive se fac în afara blocurilor *always* și *initial*. Atribuirile cu *assign* sunt atriburi continue, și nu trebuie să fie puse în interiorul blocurilor *always* (cod executat ciclic), ci în afara acestora.

Blocul *always@ edge-triggered*

În [laboratorul 2](#), 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 sensitivități 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*.

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

Exemple sensitivity list

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

Atribuirii non-blocante

În blocurile *always@* din laboratoarele precedente au fost folosite atribuirile cu *=*, numite *atriburi blocante*, deoarece se execută secvențial, ca în limbajele de programare procedurale (C, Java etc). Verilog oferă și un alt tip de atribuirii, 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 *atribuirii 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 atriburi non-blocante

```
always @(posedge sig) begin // executat pe frontul crescător al
    semnalului sig
    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).

În exemplul de mai jos, semnalul de reset este verificat **sincron**, atribuirile făcute ieșirii Q fiind **non-blocante**.

Bistabilul D - reset verificat sincron

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

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

endmodule
```

Verificarea resetului se poate realiza și în mod asincron.

Click pentru 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`.

Bistabilul D - reset verificat asincron (modul sintetizabil)

```
module D_flip_flop(output reg Q, input D, clk, rst);  
  
    always @(posedge clk or negedge rst) begin  
        if(!rst)  
            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 atribuiri la ieșirea Q sunt **blocante** în cazul în care semnalul !rst devine 1 logic sau **non-blocante** pe frontul crescător al semnalului clk. În acest caz, se obține un modul nesintetizabil.

Bistabilul D - reset verificat asincron (modul nesintetizabil)

```
module D_flip_flop(output reg Q, input D, clk, rst);  
  
    always @(posedge clk) begin  
        if(rst)  
            Q <= D;  
        end  
  
    always @(*) begin  
        if(!rst)  
            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);
  reg [2:0] state, next_state;

  // partea secventiala
  always @(posedge clk) begin
    if (reset == 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 utilităților nu vor sintetiza corect un astfel de circuit.

Exerciții

1. Se dorește proiectarea unui automat finit capabil să recunoască secvența "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 șirul "ba".
 1. **(2p)** Implementați automatul în Verilog.
 - Hint: Consultați [laboratorul 0](#) pentru diagrama de tranziție a unui astfel de automat.
 2. **(2p)** 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. **(2p)** Urmăriți diagrama de semnale și codul automatului și explicați comportamentul. Urmăriți și explicați funcționarea modulului de test.
 4. **(1p)** Testați implementarea circuitului pe placa de laborator.
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. **(3p)** 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 astfel de automat.
 2. **(1p)** Testați implementarea circuitului pe placa de laborator.
 3. **(1p)** Explicați codul numărătorului din fișierul *counter.v*.
 - Hint: Urmăriți comportarea acestuia pe diagrama de semnale.

Resurse

- [Schelet de cod](#)
- [Soluție laborator](#) (disponibilă începând cu 31.10.2018)
- [PDF laborator](#)

From:

<http://elf.cs.pub.ro/ac/wiki/> - **AC Wiki**

Permanent link:

<http://elf.cs.pub.ro/ac/wiki/lab/lab04>

Last update: **2018/10/04 12:10**

