Verilog HDL

– Curs 2 –

***Verilog HDL** reprezintă un limbaj utilizat pentru descrierea sistemelor numerice. Sistemele numerice pot fi calculatoare, componenete ale acestora sau alte structure care manipulează informație numeric.*

Verilog poate fi utilizat pentru descrierea sisemelor numerice din punct de vedere comportamental si structural:

- **Descrierea comportamentală este legată de modul în care operează sistemul şi utilizează construcții ale limbajelor traditionale de programare, de exemplu *if* sau *atribuiri*.**

- **Descrierea structurală exprimă modul în care entitățile/ componentele logice ce alcătuiesc sistemul sunt interconectate în vederea realizării comportamentului dorit.**

Structura unui Program

Limbajul Verilog descrie un sistem numeric *ca un set de module*. Fiecare dintre aceste module are o interfata cu alte module, pentru a specifica maniera in care sunt interconectate. Modulele opereaza concurent.

Modulele reprezinta parti hardware, care pot fi de la simple porti pana la sisteme complete cum ar fi un microprocesor.

O specificare comportamentala defineste comportarea unui sistem numeric (modul) folosind constructiile limbajelor de programare traditionale.

O specificare structurală exprimă comportarea unui sistem numeric (modul) ca o conectare ierarhica de submodule.

Structura unui Program

Structura unui modul este urmatoarea:

```
module <nume_modul> (<lista de porturi>);
<declaratii>
<obiecte ale modulului>
endmodule
```

<nume_modul> reprezinta un identificator care, in mod unic, denumeste modulul.

<lista de porturi> constituie o lista de porturi de intrare (input), iesire (output) sau intrare/iesire (inout), care sunt folosite pentru conectarea cu alte module.

<declaratii> specifica obiectele de tip date ca registre (reg), memorii si fire (wire), cat si constructiile procedurale ca function-s si task-s

<obiecte ale modulului> poate contine: constructii initial, constructii always, atribuiri continue sau aparitii/instante ale modulelor.

Structura unui Program

Exemplu

```
module NAND(in1, in2, out);
        input in1, in2;
        output out;
                    // instructiune de atribuire continuă
        assign out = ~(in1 & in2);
endmodule
```

Porturile in1, in2 şi out sunt etichete pe fire.

assign urmareşte în permanență eventualele modificari ale variabilelor din membrul drept, pentru reevaluarea expresiei şi pentru propagarea rezultatului în membrul stang (out).

Observație!!!!

Instrucțiunea de atribuire continuă este utilizată pentru a modela *circuitele combinaționale* la care ieşirile se modifică ca urmare a modificărilor intrărilor.

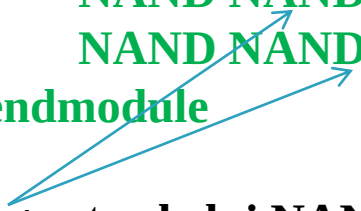
Invocarea unei instanțe este următoarea:

<nume_modul > <lista de parametri > <numele instantei> (<lista de porturi>);

<lista de parametri> are valorile parametrilor, care sunt transferate către instanță (ex. întârzierea pe o poartă).

Exemplu:

```
// Modelul structural al unei porti AND formata din doua porti NAND
                    module AND(in1, in2, out);
                        input in1, in2;
                        output out;
                        wire w1;
                        NAND NAND1(in1, in2, w1);
                        NAND NAND2(w1, w1, out);
                    endmodule
```



Acest modul are doua instanțe ale lui NAND conectate printr-un fir intern w1.

Exemplu de modul de nivel inalt, care stabileste anumite seturi de datesi care asigura monitorizarea variabilelor.

```
module test_AND;
reg a, b;
wire out1, out2;
initial begin // Datele de test
a = 0; b = 0;
#1 a = 1;
#1 b = 1;
#1 a = 0;
end
initial begin // Activarea monitorizarii
$monitor("Time=%0d a=%b b=%b out1=%b out2=%b",
$time, a, b, out1, out2);
end
// Instantele modulelor AND si NAND
AND gate1(a, b, out2);
NAND gate2(a, b, out1);
endmodule
```

- **Variabilele de tip reg stocheaza ultima valoare care le-a fost atribuita procedural.**
- **Firul, wire, nu are capacitatea de memorare. El poate fi comandat in mod continuu, de exemplu, prin instructiunea de atribuire continua assign sau prin iesirea unui modul.**

Atribuiri

- **continue** → **folosesc cuvantul cheie assign**
- **procedurale** → **<variabila reg > = <expresie>**
  unde <variabila reg> trebuie să fie un registru sau o memorie

apar *numai* în construcțiile *initial* şi *always*

Instrucțiunile din blocul construcției *initial* vor fi executate secvențial, dintre care unele vor fi întârziate de #1 cu o unitate de timp simulat.

Construcția *always* se comportă în acelaşi mod ca şi construcția *initial* cu excepția că ea ciclează la infinit (până la terminarea simulării).

- **atribuirea procedurală modifică starea unui registru, adica a logicii secvențiale**
- **atribuirea continuă este utilizată pentru a modela logica combinațională. Atribuirile continue comandă variabile de tip *wire***

Convenții lexicale

- **Limbajul este case sensitive.**
- **Numerele sunt specificate:**

<dimensiune>< format baza><numar>

<dimensiune> specifica dimensiunea constantei ca număr de *biți* (opțional)
<format baza> are un singur caracter ' urmat de unul dintre următoarele caractere b, d, o si h, care specifica baza de numeratie: binara, zecimala, octala si hexazecimala.
<numar> contine cifre, care corespund lui < format baza>.

Exemple:

```
549 // numar zecimal
'h 8FF // numar hexzecimal
'o765 // numar octal
4'b11 // numarul binar cu patru biti 0011
3'b10x // numar binar cu 3 biti, avand ultimul bit necunoscut
5'd3 // numar zecimal cu 5 ranguri
-4'b11 // complementul fata de 2, pe patru ranguri al numarului 0011 sau 1101
```

Tipuri de Date Fizice

- Variabilele **reg** stocheaza ultima valoare, care le-a fost atribuită procedural.

- Variabilele **wire** reprezintă conexiuni fizice între entități structurale cum ar fi porțile. Un fir (**wire**) nu stochează o valoare. O variabiă **wire** reprezintă numai o etichetă pe un fir.

Exemplu 1:

```
reg [0:7] A, B;
wire [0:3] Dataout;
reg [7:0] C;
```

Exemplu 2:

```
initial begin: int1
A = 8'b01011010;
B = {A[0:3] | A[4:7], 4'b0000};  // B este fortat la o valoare egala cu suma logica a primilor patru biti din A si a ultimilor patru biti din A, concatenata cu 0000. B are acum valoarea 11110000
end
```

Intr-o expresie gama de referire pentru indici trebuie sa aibe expresii constante. Un singur bit poate fi referit ca o variabila.

Exemplu:

```
reg [0:7] A, B;
B = 3;
A[0: B] = 3'b111; // ILEGAL – indicii trebuie sa fie constanti!!
```

Un argument poate fi replicat prin specificarea numărului de repetiții.

Exemple:

```
C = {2{4'b1011}};      //lui C i se asigneaza vectorul de biti: 8'b10111011
C = {{4{A[4]}}, A[4:7]};        // primii 4 biti reprezinta extensia lui A[4]
A[B] = 1'b1;                    // referirea la un singur bit este LEGALA
```

Tipuri de Date Abstracte

- **integer** reprezinta un intreg de 32 de biti cu semn
- **real** este fara semn
- **time** specifica cantitati, pe 64 de biti, care sunt folosite in conjunctie cu functia de sistem $time.

Operatori aritmetici binari: + ; - ; *; / ; + %

Operatorii relationali compara doi operanzi si intorc o valoare logica, adica TRUE (1) sau FALSE (0) : <; >; <= ; >= ; == ; != .

Operatorii logici opereaza cu operanzi logici si intorc o valoare logica, adica TRUE (1) sau FALSE (0). Sunt utilizati in instructiunile *if* si *while*. A nu se confunda cu operatorii logici Booleeni la nivel de bit.

! Negatia logica
&& AND logic
|| OR logic

Operatori la nivel de bit:

~	**Negația la nivel de bit**
&	**AND la nivel de bit**
|	**OR la nivel de bit.**
^	**XOR la nivel de bit**
~&	**NAND la nivel de bit**
~|	**NOR la nivel de bit**

~^ sau ^~ Echivalența la nivel de bit NOT XOR

{ , } Concatenarea: {A[0], B[1:7]} concateneaza bitul zero din A cu bitii 1 pana la 7 din B.

<< Deplasare la stanga:

A = A << 2; // deplaseaza A cu doi biti la stanga si forțează zero în biții eliberați.

>> Deplasare la dreapta

?: Condițional:

A = C>D ? B+3 : B-2 //semnifică faptul că dacă
//C > D, atunci valoarea lui A este B+3, altfel B-2.

Construcțiile de control

sunt utilizate în secțiunile procedurale de cod,
adică în cadrul blocurilor **initial** şi **always**

1. Selecția

instructiunea if

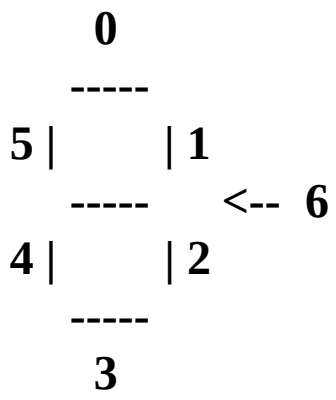
```
if (A == 4)
   begin
            B = 2;
   end
else
   begin
            B = 4;
   end
```

instrucțiunea case.

```
case (<expression>)
        <value1>: <instructiune>
        <value2>: <instructiune>
        default: <instructiune>
endcase
```

Exemplu de convertor binar-hexazecimal

7-segment encoding




```
module segment7(sin, sout);
        input [7:0] sin;
        output [6:0] sout;
        reg [6:0] sout;
  always @(sin)
    case (sin)
      8'b00000001 : sout = 7'b1111001;   // 1
      8'b00000010 : sout = 7'b0100100;   // 2
      8'b00000011 : sout = 7'b0110000;   // 3
      8'b00000100 : sout = 7'b0011001;   // 4
      8'b00000101 : sout = 7'b0010010;   // 5
      8'b00000110 : sout = 7'b0000010;   // 6
      8'b00000111 : sout = 7'b1111000;   // 7
      8'b00001000 : sout = 7'b0000000;   // 8
      8'b00001001 : sout = 7'b0010000;   // 9
      8'b00001010 : sout = 7'b0001000;   // A
      8'b00001011 : sout = 7'b0000011;   // b
      8'b00001100 : sout = 7'b1000110;   // C
      8'b00001101 : sout = 7'b0100001;   // d
      8'b00001110 : sout = 7'b0000110;   // E
      8'b00001111 : sout = 7'b0001110;   // F
      default : sout = 7'b1000000;   // 0
    endcase
endmodule
```

2. Repetiția – Instrucțiunile for, while şi repeat.

```
for(i = 0; i < 10; i = i + 1)
    begin
        $display("i= %0d", i);
    end
```

```
i = 0;
while(i < 10)
  begin
      $display("i= %0d", i);
      i = i + 1;
   end
```

```
repeat (5)
  begin
    $display("i= %0d", i);
     i = i + 1;
end
```

- **instructiunea parameter permite programatorului sa dea unei constante un nume.**

```
parameter byte_size = 8;
reg [byte_size - 1:0] A, B;
```

- **atribuirea continuă assign comandă variabile de tip wire, fiind evaluate şi actualizate ori de cate ori o intrare operand îşi modifică valoarea.**

```
assign out = ~(in1 & in2);
```

- **atribuiri procedurale blocante şi nonblocante**

evalueaza termenul din dreapta, *pentru unitatea curenta de timp*, si atribuie valoarea obtinuta termenului din stanga, la sfarsitul unitatii de timp (operator <=) .

întreaga instrucțiune este efectuată înainte de a trece controlul la următoarea instrucțiune (operator =).

Exemplu:

```
// se presupune că inițial a = 1.
// testarea atribuirii blocante

always @(posedge clk)
        begin
                a = a+1;        //în acest moment a=2
                a = a+2;        //în acest moment a=4

        end                     //rezultat final a=4

// testarea atribuirii nonblocante

always @(posedge clk)
        begin
                a <= a+1;       //în acest moment a=2
                a <= a+2;       //în acest moment a=3

        end                     //rezultat final a=3

// se folosesc vechile valori ale variabilelor, de la
// începutul unității curente de timp
```

Construcții procedurale

Blocurile **initial** şi **always** au aceeaşi construcție dar diferă prin comportare:

❖ blocurile *initial* sunt utilizate pentru inițializarea variabilelor, pentru efectuarea funcțiilor legate de aplicarea tensiunii de alimentare, pentru specificare stimulilor inițiali, monitorizare, generarea unor forme de undă;

❖ un bloc *initial* se execută o singură dată; după terminarea tuturor instrucțiunilor din blocul dat, fluxul ia sfârşit, fiind reluat odata cu simularea;

❖ blocurile *always* sunt utilizate pentru a descrie comportamentul sistemului;

❖ un bloc *always* este executat în mod repetat, într-o buclă infinită pană la terminarea simulării specificată printr-o funcție sau task de system: $finish, $stop.

❖ este important ca blocul *always* să conțină cel puțin o instrucțiune cu întârziere sau controlată de un eveniment, în caz contrar blocul se va repeta la timpul zero, blocând simularea.

Task-uri şi funcții

Task-urile sunt asemănătoare procedurilor din alte limbaje de programare.
Funcțiile se comportă ca subrutinele din alte limbaje de programare.

Excepții:

1. O funcție Verilog trebuie să se execute într-o unitate de timp simulat. Nu vor exista instrucțiuni de control al timpului: comanda întârzierii (#), comanda de eveniment (@) sau instructiunea **wait.** Un task poate conține instructiuni controlate de timp.

2. O funcție Verilog *nu* poate invoca (call, enable) un task; in timp ce un task poate chema alte task-uri şi funcții.

Definiție task:

task <nume_task >;
  <porturi argumente>
  <declaratii>
  <instructiuni>
endtask

Invocare task:

<nume_task > (<lista de porturi>);

```
module tasks;
task parity;
        input  [3:0] x;
        output z;
        z = ^ x;
endtask;

initial begin: init1
        reg r;
        parity(4'b 1011,r); // invocare task
        $display("p= %b", r);
end
 endmodule
```

```
task factorial;
        input [3:0] n;
        output [31:0]  outfact;
        integer  count;
        begin
            outfact = 1;
            for (count = n; count>0; count = count-1)
                outfact = outfact * count;
end
endtask
```

Scopul unei *funcții* este acela de a returna o valoare, care urmează să fie folosită într-o expresie.

Definiție funcție:

```
function <gama sau tipul> <nume_functie>;
        < porturi argumente>
        <declaratii>
        <instructiuni>
endfunction
```

```
module functions;
function  parityf;
        input  [3:0] x;
        parityf = ^ x;
endfunction;

initial begin: init1
        reg r;
        parityf(4'b 1011,r);
                // invocarea task-ului
        $display("p= %b", r);
end
 endmodule
```

```
function [31:0] factorial;
        input [3:0] operand;
        reg [3:0] i;
        begin
                factorial = 1;
                for (i=2; i<=operand; i=i+1)
                        factorial = i * factorial;

        end
endfunction
```

Controlul sincronizării/Timing-ului.

Limbajul Verilog oferă trei tipuri explicite de control al sincronizării, atunci când urmează să apară instrucțiuni procedurale.

- **comanda întârzierii** în care o expresie specifică durata de timp între prima apariție a instrucțiunii şi momentul în care ea se execută.
- **expresia eveniment**, care permite execuția instrucțiunii.
- instrucțiunea **wait,** care asteaptă modificarea unei variabile specifice.

Timpul de simulare poate progresa *numai* în una din următoarele situații:

1. specificarea întârzierii pe poartă sau fir;
2. un control al întârzierii, introdus prin simbolul #;
3. un eveniment de control, introdus prin simbolul @;
4. instrucțiunea **wait**.

Controlul întârzierii:

#10 A = A + 1; ⟹ specifică o întârziere de 10 unități de timp înainte de a executa instrucțiunea de asignare procedurală.

Apariția unui eveniment cu nume:

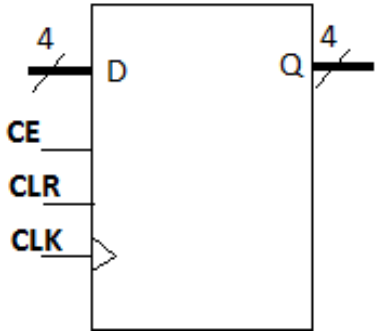
```
@r begin                          // controlat printr-o modificare a valorii in registrul r
    A = B&C;
end

@(posedge clock2) A = B&C;        // controlat de frontul pozitiva al lui clock2

@(negedge clock3) A = B&C;        // controlat de frontul negativ al lui clock3

forever @(negedge clock)          // controlat de frontul negativ
    begin
       A = B&C;
    end
```

Instrucțiunea wait

permite întârzierea unei instrucțiuni procedurale sau a unui bloc, până ce condiția specificată devine adevarată:

```
wait (A == 3)
begin
    A = B&C;
end
```

Exemplul 1: Implementarea unui registru de tipul D



D	CE	CLR	CLK	Q (t)
x	x	1	x	0
x	0	0	↓	Q(t-1)
y	1	0	↓	y

```
module ffd(d, ce, clr, clk, q);
        input               ce, clr, clk;
        input               [3:0] d;
        output    [3:0] q;
        reg                 [3:0] q;
 always @(clr)
        q <= 0;

always @(posedge clk)
begin

        if (ce)
                q <= d;

end
endmodule
```

Exemplul 2: Instanţierea unui modul

```
module modul_de_instanţiat(a, b, c, d);

        input a, b, c;
        output d;
        always @(a, b, c)begin

                d = a xor b;
                a = b xor c;

        end;

endmodule

module principal;

        wire dout;          //ieşirile circuitului final
        reg in1, in2, in3;  //intrările circuitului

        modul_de_instanţiat test(in1, in2, in3, dout);

endmodule
```

Exemplul 3: Testarea unui modul

```
module sumator(a, b, cin, sum, cout);
        input a, b, cin;
        output cout, sum;

        always @(a or b or cin)

                {cout, sum} = a + b + cin;
endmodule

module simulare
        reg     a, b, cin;
        wire    cout, sum;
        sumator inst1(a, b, cin, sum, cout);        //instanţiere

        initial begin
                a = 0; b = 0; cin = 0;
                #10  a=0; b=0; cin=1;
                #10  a=1; b=1; cin=1;
                #10  $finish
        end

        initial
            $monitor($time, "Suma este %b iar transportul este %b", sum, cout);

endmodule
```

Exemplul 4: Un sumator descris structural

```
module semiSumator(sum, cout, a, b);
        input a, b;
        output cout, sum;

        xor #2(sum, a, b);
        and #2(cout, a, b);

endmodule
```

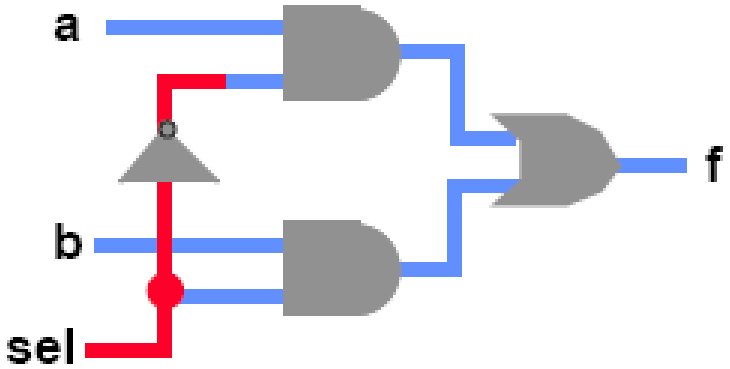
```
module testSumator (a, b, cout, sum);
        input sum, cout;
        output a, b;
        reg a, b;

        initial begin
                $monitor($time, "a = %b, b = %b, sum = %b,
                        cout = %b", a, b, sum, cout);
                a = 0, b = o;
                #10 b = 1;
                #10 a = 1;
                #10 b = 0;
                #10 $finish;
                end
endmodule
```

```
module testBench;
        wire su, co, a, b;

        semiSumator add(su, co, a, b);
        testSumator ta(a, b, su, co);

endmodule
```

Observație: cu ajutorul unui modul testBench se conectează modul design în care s-a proiectat cu modulul de testare.

Exemplul 5: Un multiplexor descris structural

```
module mux( f, a, b, sel);
        input a, b, sel;
        output f;
        wire nsel, f1, f2;

        and     #5 g1(f1, a, nsel), g2(f2, b, sel);
        or      #5 g3(f, f1, f2);
        not     g4 (nsel, sel)
endmodule
```



Exemplul 6: Un comparator mai mic sau egal

```
module comparator_LEQ(a, b ,c);

        parameter Width = 8;
        input [Width-1:0] a,b;
        output c;

        assign c = (a<b))?1:0;

endmodule
```

Exemplul 7: Un registru pe 8 biți

```
module registru(rout, rin, clear, load, clock);

        parameter Width = 8;
        output [Width-1:0] rout;
        reg [Width-1:0] rout;
        input [Width-1:0] rin;
        input clear, load, clock;

        always @ (posedge clock)
                if (~clear)
                        rout <= 0;
                else    if (~load)
                        rout <= rin;

endmodule
```