Verilog HDL

– continuare –

2. Repetiţia – Instrucţiunile 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 instrucţiune este efectuată înainte de a trece controlul la următoarea instrucţiune (operator =).

Exemplu:

// se presupune că iniţial **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 unităţii curente de timp

Construcţii procedurale

Blocurile **initial** şi **always** au aceeaşi construcţie dar diferă prin comportament:

❖ blocurile *initial* sunt utilizate pentru iniţializarea variabilelor, pentru efectuarea funcţiilor legate de aplicarea tensiunii de alimentare, pentru specificarea stimulilor iniţiali, monitorizare, generarea unor forme de undă;

❖ un bloc *initial* se execută o singură dată; după terminarea tuturor instrucţiunilor 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 funcţie sau task de system: $finish, $stop.

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

Task-uri şi funcţii

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

Excepţii:

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

2. O funcţie Verilog *nu* poate invoca (call, enable) un task, în timp ce un task poate apela alte task-uri şi funcţii.

Definiţie 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 *funcţii* este acela de a returna o valoare, care urmează să fie folosită într-o expresie.

Definiţie funcţie:

```
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 instrucţiuni procedurale.

- **comanda întârzierii** în care o expresie specifică durata de timp între prima apariţie a instrucţiunii şi momentul în care ea se execută.
- **expresia eveniment**, care permite execuţia instrucţiunii.
- instrucţiunea **wait,** care asteaptă modificarea unei variabile specifice.

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

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

Controlul întârzierii:

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

Apariţia unui eveniment cu nume:

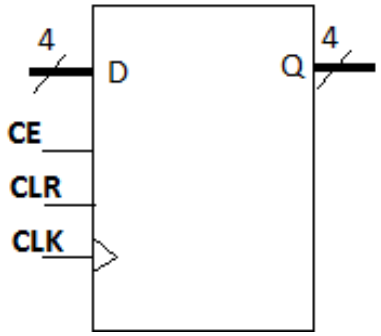
```
@r begin                          // controlat printr-o modificare a valorii in registrul r
    A = B&C;
end

@(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
```

Instrucţiunea wait

permite întârzierea unei instrucţiuni procedurale sau a unui bloc, până ce condiţia specificată devine adevărată:

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

Exemplul 1: Implementarea unui bistabil 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, e);

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

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

        end;

endmodule

module pentru_test;

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

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

endmodule
```

Exemplul 3: Testarea unui modul

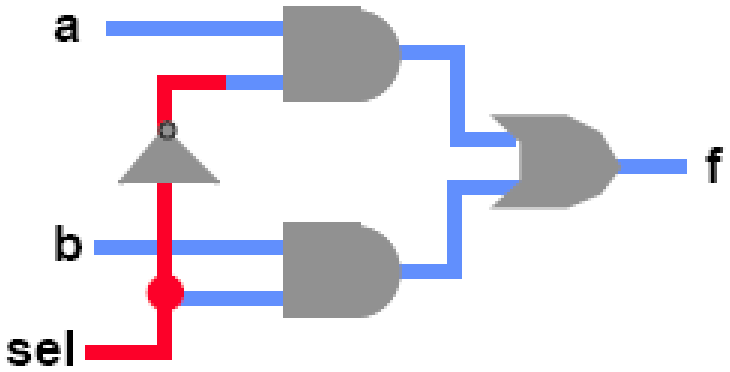
```
module sumator(a, b, cin, sum, cout);
        input a, b, cin;
        output cout, sum;
        reg cout, sum;
        always @(a or b or cin)

                {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 semi-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);
        wire sum, cout;
        reg a, b;

        semiSumator inst1 (sum, cout, a, b);
        initial begin
                $monitor($time, "a = %b, b = %b, sum = %b,
                        cout = %b", a, b, sum, cout);
                a = 0, b = 0;
                #10 b = 1;
                #10 a = 1;
                #10 b = 0;
                #10 $finish;
                end
endmodule
```

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
```



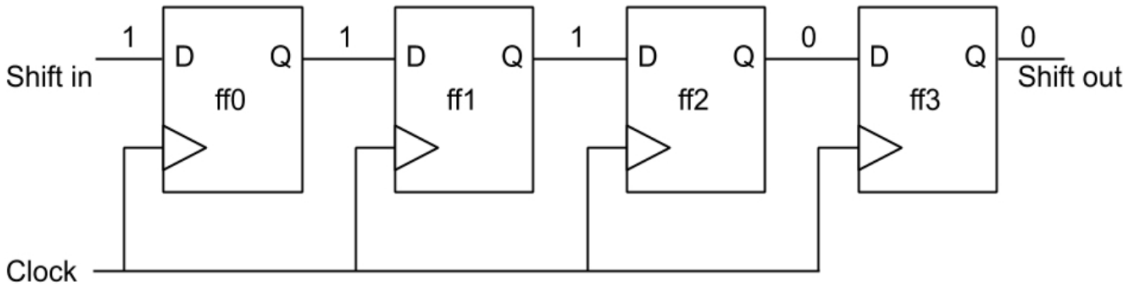
Exemplul 6: Un comparator mai mic sau egal

```
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 biţi

```
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
```

Exemplul 8: Shift register in Verilog




```
module shiftReg4(shift_in, clock, shift_out)
  input shift_in, clock;
  output shift_out;
  reg bit0, bit, bit2, bit3;

assign shift_out = bit3;

always @(posedge clock) begin
  bit3 <= bit2;
  bit2 <= bit1;
  bit1 <= bit0;
  bit0 <= shift_in;
end

endmodule
```