VERILOG HDL

prof. dr. ing. Adrian Petrescu
s.l. dr. ing. Nirvana Popescu

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. In cele ce urmează accentual se va pune pe descrierea sisemelor numerice la nivelul transferurilor între registre (RTL – Register Transfer Language).

- Verilog VHDL (~ limbajul C)
- VHDL (Very High speed integrated circuit hardware Description Language) (~ ADA)

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 vedeea realizării comportamentului dorit.

Primul program Verilog.

```
//Dan Hyde
// Un prim model numeric in Verilog
module simple;
// Exemplu simplu la nivel RTL, pentru a demonstra Verilog.
/* Registrul A este incrementat cu unu. Apoi primii patru biti ai registrului B sunt
incarcati cu valorile negate ale ultimilor patru biti din A. C reprezinta o reducere “and” //
a ultimilor doi biti din A.*/
//declara registrele si bistabilele
reg [0:7] A, B;
reg C;
// Cele doua constructii “initial” si always” opereaza concurent
initial begin: stop_at
// va opri executia dupa 20 unitati de simulare.
#20; $stop;
end
// Aceste instructiuni sunt executate la timpul simulat 0 ( nu exista #k).
initial begin: Init
// Initializeaza registrul A. Celelalte registre au valori “x”.
A = 0;
// Antetul de afisare.
$display("Time A B C");
// Afiseaza valorile A, B, C in cazurile in care acestea se modifica.
$monitor(" %0d %b %b %b", $time, A, B, C);
end
//main_process va cicla pana la terminarea simularii.
always begin: main_process
// #1 insemna executia dupa o unitate de timp de simulare.
#1 A = A + 1;
#1 B[0:3] = ~A[4:7]; // ~ este operatorul "not" la nivel de bit
#1 C = &A[6:7]; // reducerea "and" a ultimilor doi biti ai registrului
end
endmodule
```

In **module simple** au fost declarate doua registrea **A** si **B** de cate 8 biti si un registru **C**, de un bit sau bistabil. In interiorul modulului, o constructie “**always**” si doua constructii “**initial**” descriu **firele de control**, care se executa in acelasi timp sau **concurent.** In constructia **initial** instructiunile sunt executate secvential ca in C sau in alte limbaje imperative traditionale. Constructia **always** este asemanatoare cu constructia **initial** cu exceptia ca aceasta cicleaza continuu, cat timp opereaza simulatorul.

Notatia #**1** semnifica faptul ca instructiunea care urmeaza se va executa cu o

intarziere de o unitate de timp simulat. De aceea, firul de control, provocat de catre prima constructie
initial va intarzia, cu 20 unitati de timp simulat, chemarea task-ului de sistem **$stop** si va opri simularea.
Task-ul de sistem **$display** permite proiectantului sa tipareasca mesajele in maniera in care **printf** o face in limbajul C. La fiecare unitate de timp simulat, cand una dintre variabilele listate isi modifica valoarea, task- ul de sistem **$monitor** tipareste un mesaj. Functia de sistem **$time** intoarce valoarea curenta a timpului simulat.
Mai jos se prezinta iesirea Simulatorului VeriWell: (A se vedea Sectiunea 3, referitoare la utilizarea Simulatorului VeriWell).

```
Time A B C
0 00000000 xxxxxxxx x
1 00000001 xxxxxxxx x
2 00000001 1110xxxx x
3 00000001 1110xxxx 0
4 00000010 1110xxxx 0
5 00000010 1101xxxx 0
7 00000011 1101xxxx 0
8 00000011 1100xxxx 0
9 00000011 1100xxxx 1
10 00000100 1100xxxx 1
11 00000100 1011xxxx 1
12 00000100 1011xxxx 0
13 00000101 1011xxxx 0
14 00000101 1010xxxx 0
16 00000110 1010xxxx 0
17 00000110 1001xxxx 0
19 00000111 1001xxxx 0
Stop at simulation time 20
```

Acest program, cat si rezultatele simularii trebuie studiate cu atentie. Structura programului este tipica pentru programele Verilog, care vor fi scrise in acest curs: o constructie **initial,** pentru a specifica durata simularii, o alta constructie **initial**, pentru initializarea registrelor si pentru a specifica registrele care se monitorizeaza si o constructie **always**, pentru sistemul numeric simulat. De observat ca toate instructiunile din cea de-a doua constructie **initial** sunt efectuate la timpul time = 0, deoarece nu sunt prevazute instructiuni de intarziere: #<**integer**>.

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. De regula, un modul se plaseaza intr-un fisier, fara ca aceasta sa fie o cerinta obligatorie. Modulele opereaza concurent.

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

Modulele pot fi specificate, fie comportamental, fie structural (sau o combinatie a celor doua).

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

O **specificare structural** exprima comportarea unui sistem numeric (modul) ca o conectare ierarhica de submodule. La baza ierarhiei componentele trebuie sa fie primitive sau sa fie specificate comportamental.

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.
Un modul nu este chemat *niciodata*. Un modul are o instanta la inceputul unui program si este prezent pe intreaga durata a programului. O instanta a unui modul Verilog este utilizata ca model al unui circuit hardware la care se presupune ca nu se efectueaza modificari de cablaj. La fiecare instanta a modulului, acesteia i se da un nume.

De exemplu **NAND1** şi **NAND2** sunt nume de instanțe ale porții **NAND**. În acest exemplu este data o specificare comportamentala a unui modul **NAND**. Iesirea **out** este negatia produsului **and** al intrarilor **in1** si **in2.**

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

Porturile **in1**, **in2** si **out** sunt etichete pe fire. Atribuirea continua **assign** urmareste in ermanenta eventualele modificari ale variabilelor din membrul drept, pentru reevaluarea expresiei si pentru propagarea rezultatului in membrul stang (**out**).

Instructiunea de atribuire continua este utilizata pentru a modela **circuitele combinationale** la care iesirile se modifica ca urmare a modificarilor intrarilor. Specificarea structurala a unui modul **AND,** obtinuta ca urmare a conectarii iesirii unui **NAND** la cele doua intrari ale altui **NAND**.

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

Acest modul are doua instante ale lui **NAND**, numite **NAND1** si **NAND2**, conectate printr-un fir intern **w1**

Forma generala pentru invocarea unei instante este urmatoarea:

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

<**lista de parametri**> are valorile parametrilor, care sunt transferate catre instanta.
Un exemplu de parametru transferat ar fi intarzierea pe o poarta.

Urmatorul modul reprezinta un modul de nivel inalt, care stabileste anumite seturi de date si care asigura monitorizarea variabilelor.

```
module test_AND;
// Modul de nivel inalt pentru testarea altor doua module.
```

```
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
```

De notat ca valorile **a** si **b** trebuie mentinute pe toata durata operarii. De aceea este necesara utilizarea unor registre de cate un bit.

- 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. Daca firele de intrare nu sunt conectate, ele vor lua valoarea **x** – necunoscuta.

Atribuirile pot fi de continue sau procedurale.
Atribuirile continue folosesc cuvantul cheie **assign** in timp ce atribuirile procedurale au forma:
<variabila reg > = <expresie>
unde **<variabila reg>** trebuie sa fie un registru sau o memorie.

Atribuirile procedurale pot aparea *numai* in constructiile **initial** si **always.**
Instructiunile din blocul primei constructii **initial** vor fi executate secvential, dintre care unele vor fi intarziate de **#1** cu o unitate de timp simulat.

Constructia **always** se comporta in acelasi mod ca si constructia **initial** cu exceptia ca ea cicleaza la infinit (pana la terminarea simularii).

Verilog face o importanta distinctie intre atribuirile procedurale si atribuirea continua **assign.**
Atribuirile **procedurale** modifica starea unui registru, adica a logicii secventiale, in timp ce atribuirea **continua** este utilizata pentru a modela **logica combinationala**. Atribuirile continue comanda variabile de tip **wire** si sunt evaluate si actualizate atunci cand un operand de intrare isi modifica valoarea.

Conventii Lexicale.

Comentariile sunt specificate prin // la inceputul liniei, fiind valabile pentru acea linie. Comentariile pe mai multe linii incep cu /* si se termina cu */.

Cuvintele cheie, de exemplu, **module**, sunt rezervate si utilizeaza literele mici ale alfabetului. Limbajul este case sensitive. Spatiile sunt importante prin aceea ca delimiteaza obiectele in limbaj.
Numerele sunt specificate sub forma traditionala ca o serie de cifre cu sau fara semn, in maniera urmatoare:

<dimensiune>< format baza><numar>

unde <**dimensiune**> contine cifre *zecimale*, care specifica dimensiunea constantei ca numar de *biti*.
Campul <**dimensiune**> este optional.
Campul <**format baza**> are un singur caracter **'** urmat de unul dintre urmatoarele caractere **b**, **d**, **o** si **h**, care specifica baza de numeratie: binara, zecimala, octala si hexazecimala.
Campul <**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

1. Tipuri de Date Fizice.

Intrucat scopul limbajului Verilog este acela de a modela hardware-ul numeric, tipurile primare de date vor fi destinate modelarii registrelor (**reg**) si firelor (**wire**).

- Variabilele **reg** stocheaza ultima valoare, care le-a fost atribuita procedural.
- Variabilele **wire** reprezinta conexiuni fizice intre entitati structurale cum ar fi portile. Un fir (**wire**) nu stocheaza o valoare. O variabila **wire** reprezinta numai o eticheta pe un fir.

Obiectele **reg** si **wire** , care reprezinta date, pot lua urmatoarele valori:

0 valoarea logica zero sau fals,
1 unu logic sau adevarat,
x valoare logica necunoscuta,
z impedanta ridicata a unei porti “tristate”.

La inceputul simularii variabilele **reg** sunt initializate la valoarea **x**. Oricare variabila **wire** neconectata la ceva are valoarea **x**.
In declaratii se poate specifica dimensiunea unui registru sau a unui fir. De exemplu, declaratiile:

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

specifica registrele **A** si **B** ca avand 8 biti, cu cel mai semnificativ bit avand indicele zero, in timp ce registrul **C**, de opt biti, are indicele sapte, pentru cel mai semnificativ bit. Firul **Dataout** are 4 biti.

Bitii unui registru sunt specificati prin notatia:**[<start-bit>:<end-bit>]**.
De exemplu, in cea de-a doua atribuire procedurala, din secventa:

```
initial begin: int1
A = 8'b01011010;
B = {A[0:3] | A[4:7], 4'b0000};
end
```

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. Parantezele **{}** specifica faptul ca bitii a doua sau ai mai multor argumente, separati prin virgule, sunt concatenati impreuna.

Un argument poate fi replicat prin specificarea numarului de repetitii sub forma:
{numar_repetitii{exp1, exp2, ... , expn}}
Iata cateva 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].
```

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

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

A[B] = 1'b1; // Referirea la un singur bit este LEGALA.

Aceasta cerinta stricta privind valorile constante pentru indici, la referirea unui registru, este impusa de faptul ca se doreste ca expresia sa fie realizabila fizic, in hardware.

Memoriile sunt specificate ca vectori de registre.
De exemplu **Mem** are 1K cuvinte de cate 32 de biti.

reg [31:0] Mem [0:1023];

Notatia **Mem[0]** asigura referirea la cuvantul cu adresa zero din memorie. Tabloul de indici pentru o memorie (vector de registre) poate fi un registru. *De remarcat faptul ca, in Verilog, o memorie nu poate fi accesata la nivel de bit.* Pentru a realiza aceasta operatie, cuvantul din memorie trebuie transferat mai intai intr-un registru temporar.

2. Tipuri de Date Abstracte.

Intr-un model de hardware, pe langa variabilele, care modeleaza hardware-ul, se gasesc si alte tipuri de variabile. De exemplu, proiectantul doreste sa foloseasca o variabila **integer**, pentru a contoriza numarul de aparitii ale unui eveniment. Pentru comoditatea proiectantului, Verilog HDL poseda mai multe tipuri de date carora nu le corespund realizari hardware. Aceste tipuri de date includ **integer**, **real** si **time**.

Tipurile de date **integer** si **real** se comporta ca si in alte limbaje, de exemplu C.
Trebuie subliniata faptul ca variabila **reg** este fara semn, in timp ce variabial **integer** reprezinta un intreg de 32 de biti cu semn. Acest fapt are o mare importanta la scadere. Variabila **time** specifica cantitati, pe 64 de biti, care sunt folosite in conjunctie cu functia de sistem **$time.** Sunt acceptate tablouri de variabile **integer** si **time** , dar nu de **reali**. In Verilog **nu** sunt permise tablouri multidimensionale Iata cateva exemple:

integer Count; // intreg simplu, cu semn, pe 32 de biti.
integer K[1:64]; // un tablou de 64 intregi.
time Start, Stop; // doua variabile timp de 64 de biti.

Operatori

1. Operatori Aritmetici binari.

Operatorii aritmetici binari opereaza cu doi operanzi. Operanzii de tip **reg** si **net** sunt tratati ca fara semn. Operanziide tip **real** si **integer** pot avea semn. Daca un bit oarecare, al unui operand, este necunoscut (‘**x**’), atunci rezultatul este necunoscut.
Operator Nume Comentarii
+ Adunare
- Scadere
* Inmultire / Impartire Impartirea cu zero furnizeaza un **x**, necunoscut
% Modul

2. Operatori Relationali.

Operatorii relationali compara doi operanzi si intorc o valoare logica, adica TRUE (1) sau FALSE (0). Daca un bit oarecare este necunoscut, relatia este ambigua si rezultatul este necunoscut.

> Mai mare decat
>= Mai mare decat sau egal
< Mai mic decat
<= Mai mic decat sau egal
== Egalitate logica
!= Inegalitae logica

3. Operatori Logici

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. De exemplu, ! este un NOT logic, iar ~ este NOT la nivel de bit. Primul neaga, ex., !(5 == 6) este TRUE. Al doilea complementeaza bitii, de exemplu: ~{1,0,1,1} este 0100.

! Negatia logica
&& AND logic
|| OR logic

4. Operatori la nivel de bit.

Operatorii la nivel de bit actioneaza pe bitii operandului sau operanzilor. De exemplu, rezultatul lui A & B este AND pentru bitii corespunzatori din A si B. Operand pe un bit necunoscut, se obtine un bit necunoscut (x) in valoarea rezultatului. De exemplu, AND-ul unui **x** cu un FALSE este un **x**. OR-rul unui **x** cu un TRUE este un TRUE.

~ Negatia 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 ^~ Echivalenta la nivel de bit NOT XOR

5. Alti Operatori.

=== Egalitatea Case. Comparatia la nivel de bit include compararea valorilor **x** si **z**. Pentru egalitate, toti bitii trebuie sa se potriveasca. Intoarce TRUE sau FALSE.

!== Inegalitatea Case Comparatia la nivel de bit include compararea valorilor **x** si **z**. Oricare diferenta intre bitii comparati produce o inegalitate. Intoarce TRUE sau FALSE.

{ , } Concatenarea. Reuneste bitii separati, furnizati de doua sau mai multe expresii separate prin virgule, de exemplu:
{A[0], B[1:7]} concateneaza bitul zero din A cu bitii 1 pana la 7 din B.

<< Deplasare la stanga. Pozitiile din dreapta eliberate se incarca cu zero, de exemplu:
A = A << 2; // deplaseaza A cu doi biti la stanga si forteaza zero in bitii eliberati.

>> Deplasare la dreapta Pozitiile din stanga eliberate se incarca cu zero

?: Conditional Atribuie una dintre cele doua valori, in functie de conditie, de exemplu:
A = C>D ? B+3 : B-2 //semnifica faptul ca daca
//C > D, atunci valoarea lui A este B+3, altfel B-2.

6. Precedenta operatorilor.

Precedenta operatorilor este prezentata mai jos: Varful tabelei corespunde celei mai ridicate precedente, in timp ce baza corespunde celei mai coborate precedente.
Operatorii de pe aceeasi linie au aceeasi precedenta si asociaza termenul stang celui drept intr-o expresie. Parantezele sunt utilizate pentru a modifica precedenta sau pentru a clarifica situatia. Pentru a usura intelegerea, se recomanda utilizarea parantezelor.
Operatori unari: ! & ~& | ~| ^ ~^ + - (cea mai ridicata precedenta)

* / %
+ -
<< >>
< <= > > =
== != === ~==
& ~& ^ ~^
| ~|
&&
||
?:

Constructiile de Control.

Verilog poseda o bogata colectie de instructiuni de control, care pot fi utilizate in sectiunile procedurale de cod, adica in cadrul blocurilor **initial** si **always**. Cele mai multe sunt familiare programatorului, care cunoaste limbajele traditionale de programare, cum ar fi limbajul C.
Principala diferenta consta in aceea ca in locul parantezelor **{ },** din limbajul C, Verilog utilizeaza **begin** si **end**. In Verilog, parantezele **{ }** sunt utilizate pentru concatenarea sirurilor de biti. Deoarece cei mai multi utilizatori sunt familiarizati cu limbajul C, urmatoarele subsectiuni vor contine numai cate un exemplu din fiecare constructie.

1. Selectia – Instructiunile if si case.

Instructiunea **if** este usor de utilizat:

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

Spre deosebire de instructiunea **case,** din C, nu este necesară instrucțiunea **break.** Se selecteaza prima **<value>,** care se potriveste cu valoarea lui **<expression>**, fiind executata instructiunea asociata. Apoi controlul este transferat dupa **endcase**.

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

In urmatorul exemplu se verifica valoarea semnalului de 1 bit .

```
case (sig)
1'bz: $display("Semnal flotant");
1'bx: $display("Semnal necunoscut");
default: $display("Semnalul este %b", sig);
endcase
```

```
module segment7(sin, sout);   //exemplu de convertor binar-hexazecimal
          input [7:0] sin;
          output [6:0] sout;
          reg [6:0] sout;
// 7-segment encoding
//        0
//       -----
//  5 |       | 1
//       -----      <--  6
//  4 |       | 2
//       -----
//        3

  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. Repetitia – Instructiunile for, while si repeat.

Instructiunea **for** este foarte apropiata de instructiunea **for** din C, cu exceptia ca operatorii ++ si – nu sunt prezenti in Verilog. De aceea, se va folosi **i = i + 1**.

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

Instructiunea **while** opereaza in forma normala:

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

Instructiunea **repeat** repeta urmatorul bloc de un numar fixat de ori, in exemplul urmator - de 5 ori:

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

3. Alte Instructiuni

- **instructiunea parameter**

Instructiunea parametru permite programatorului sa dea unei constante un nume. In mod tipic se utilizeaza pentru a specifica numarul de biti ai unui registru sau intarzierile. De exemplu, se pot parametriza declaratiile unui model:

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

- **atribuirea continuă**

Atribuirea continua comanda variabile de tip **wire**, fiind evaluate si actualizate ori de cate ori o intrare operand isi modifica valoarea. In cele ce urmeaza valorile de pe firele **in1** si **in2** sunt inmultite pentru a comanda iesirea **out**. Cuvantul cheie **assign** este utilizat pentru a face distinctia intre atribuirea continua si atribuirea procedurala. Instructiunea urmatoare efecteaza produsul semnalelor de pe firele **in1** si **in2,** neaga rezultatul si comanda continuu firul **out**.

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

- **atribuiri procedurale blocante si nonblocante**

Limbajul Verilog are doua forme pentru instructiunea de atribuire procedurala: **blocanta** si **nonblocanta**. Cele doua forme se deosebesc prin operatorii de atribuire = si <=.

Instructiunea de atribuire blocanta (operatorul =) actioneaza ca si in limbajele traditionale de programare. Intreaga instructiune este efectuata inainte de a trece controlul la urmatoarea instructiune.

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

Exemplu:

```
//se presupune ca initial a = 1.
// testarea atribuirii blocante

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

        end                     //rezultat final a=4

// testarea atribuirii nonblocante

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

        end                     //rezultat final a=3
```

Efectul este acela ca, pentru toate atribuirile nonblocante se folosesc vechile valori ale variabilelor, de la inceputul unitatii curente de timp, pentru a asigna registrelor noile valori, la sfarsitul unitatii curente de timp. Aceasta reprezinta o reflectarea a modului in care apar unele transferuri intre registre, in unele sisteme hardware.

Construcții procedurale

Limbajul Verilog include o varietate de construcții de pogramare care permit descrierea unui sistem din punct de vedere comportamental, fără a apela la o descriere logică detaliată. Un modul Verilog poate conține un numar oarecare de proceduri care operează în paralel. Fiecare dintre aceste fluxuri paralele de execuție poartă numele de fir sau procedură. Acestea pot fi lansate, repetate sau pot astepta apariția unei condiții.

Fiecare instrucțiune initial sau always care apare într-un modul lansează un nou flux. Toate fluxurile se lansează imediat, la timpul zero al simulării. Odată lansate, fiecare flux evoluează independent, executând istrucțiuni sau asteptând apariția unei condiții date.

Blocurile initial şi always au aceeasi 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 si Functii

Task-urile sunt asemanatoare procedurilor din alte limbaje de programare, de exemplu, task-urile pot avea zero sau mai multe argumente si nu intorc o valoare. Functiile se comporta ca subrutinele din alte limbaje de programare.

Exceptii:
1. O functie Verilog trebuie sa se execute intr-o unitate de timp simulat. Nu vor exista instructiuni de control al timpului: comanda intarzierii (#), comanda de eveniment (@) sau instructiunea **wait.** Un task poate contine instructiuni controlate de timp.
2. O functie Verilog *nu* poate invoca (call, enable) un task; in timp ce un task poate chema alte task-uri si functii.
Definitia unui task este urmatoarea:

```
task <nume_task >; // de observat ca nu exista o lista de parametri sau ().
<porturi argumente>
<declaratii>
<instructiuni>
endtask
```

O invocare a unui task este de forma urmatoare:

```
<nume_task > (<lista de porturi>);
```

unde **<lista de porturi>** este o lista de expresii, care corespund prin pozitie lui **<porturi argumente>** din definitie. Porturi argumente, in definitie pot fi: **input**, **inout** sau **output**. Deoarece **<porturi argumente>** din definitia task-ului arata ca si alte declaratii, programatorul trebuie sa fie atent, in ceea ce priveste adaugarea declaratiilor la inceputul task-ului.

Exemplul1:

```
// Testarea task-urilor si a functiilor
module tasks;
task parity;
        input  [3:0] x;
        output z;

        z = ^ x;
endtask;

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

endmodule
```

Exemplul 2:

```
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
```

Parametrii **input** si **inout** sunt transferati prin valoare catre task, iar parametrii **output** si **inout** sunt transferati inapoi prin valoare, catre constructia care a invocat task-ul. Chemarea prin referire nu este disponibila. Alocarea tuturor variabilelor este statica. Astfel, un task se poate autoinvoca, dar la fiecare invocare se va folosi aceeasi memorie, ceea ce inseamna ca variabilele locale *nu* sunt fortate in stiva. Deoarece firele concurente pot invoca acelasi task, programatorul trebuie sa fie atent la natura statica a alocarii spatiului de memorie, pentru a evita suprascrierea spatiului partajat de memorie.

Scopul unei *functii* este acela de a returna o valoare, care urmeaza sa fie folosita intr-o expresie.

Definitia unei functii este urmatoarea:

```
function <gama sau tipul> <nume_functie>; //nu exista lista de parametric sau ()
< porturi argumente>
<declaratii>
<instructiuni>
endfunction
```

unde **<gama/range** sau **tipul/type>** este tipul rezultatelor returnate expresiei, care a chemat functia. In interiorul functiei, trebuie sa se asigneze o valoare numelui functiei. Mai jos este prezentata o functie, care este similara task-ului de mai sus.

Exemplul 1:

```
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
```

Exemplul 2:

```
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 Sincronizarii/Timing-ului.

Limbajul Verilog ofera trei tipuri explicite de control al sincronizarii, atunci cand urmeaza sa apara instructiuni procedurale.

- **comanda intarzierii** in care o expresie specifica durata de timp intre prima aparitie a instructiunii si momentul in care ea se executa.
- **expresia eveniment**, care permite executia instructiunii.
- instructiunea **wait,** care asteapta modificarea unei variabile specifice.

Verilog constituie un **simulator bazat pe evenimente in timp discret**, adica evenimentele sunt planificate la momente discrete de timp si plasate intr-o coada de asteptare, ordonata in timp. Cel mai timpuriu eveniment se afla in fruntea cozii de asteptare, iar evenimentele ulterioare sunt dupa el. Simulatorul inlatura toate evenimentele pentru timpul curent de simulare si le prelucreaza. In timpul prelucrarii, pot fi create mai multe evenimente si plasate in locul corespunzator, in coada pentru prelucrarea ulterioara. Cand toate evenimentele pentru timpul curent au fost procesate, simulatorul trece la urmatorul pas in timp si prelucreaza urmatoarele evenimente din fruntea cozii. Daca nu exista controlul sincronizarii, timpul de simulare nu evolueaza.

Timpul de simulare poate progresa *numai* in una din urmatoarele situatii:
1. specificarea intarzierii pe poarta sau fir;
2. un control al intarzierii, introdus prin simbolul #;
3. un eveniment de control, introdus prin simbolul @;
4. instructiunea **wait**.

Ordinea executiei evenimentelor in acelasi interval de timp/ceas este impredictibila.

1. Controlul Intarzierii (#)

O expresie de **control al intarzierii** specifica durata de timp intre prima aparitie a instructiunii si momentul executiei acesteia. De exemplu**:**

```
#10 A = A + 1;
```

specifica o intarziere de 10 unitati de timp inainte de a executa instructiunea de asignare procedurala. Simbolul # poate fi urmat de o expresie cu variabile.

2. Evenimente.

Executia unei instructiuni procedurale poate fi amorsat de o schimbare de valoare pe un fir sau intr-un registru sau de aparitia 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
```

Formele, care folosesc **posedge** si **negedge** trebuie sa fie urmate de o expresie de un bit, de regula un semnal de ceas. La o tranzitie de la 1 la 0 (sau necunoscut), se detecteaza un **negedge.** Un **posedge** este detectat la o tranzitie de la 0 la 1 (sau necunoscut).

Verilog furnizeaza, de asemenea, facilitati pentru a denumi un eveniment si apoi sa amorseze aparitia acelui eveniment. Mai intai trebuie sa se declare acel eveniment:

```
event event6;
```

Pentru a amorsa un eveniment se va folosi simbolul –>:

```
–> event6;
```

Pentru a controla un bloc de cod, se va folosi simbolul @, dupa cum se arata mai jos:

```
@(event6) begin
<cod procedural oarecare >
end
```

Se va presupune ca evenimentul apare intr-un fir de control, adica concurent, iar codul de control se afla in alt fir. Mai multe evenimente pot fi sumate logic in interiorul parantezelor.

3 Instructiunea wait.

Instructiunea **wait** permite intarzierea unei instructiuni procedurale sau a unui bloc, pana ce conditia specificata devine adevarata:

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

Diferenta intre comportamentul unei instructiuni **wait** si un eveniment este aceea ca instructiunea **wait sesizeaza nivelul,** in timp ce un **@(posedge clock);** este amorsat de catre o **tranzitie a semnalului** sau **sesizeaza frontul**.

4 Instructiunile fork si join.

Folosind constructiile **fork** si **join**, Verilog permite existenta mai multor fire de control in cadrul constructiilor **initial** si **always**. De exemplu, pentru a avea trei fire de control, se ramifica (**fork**) controlul in trei si apoi fuzioneaza (**join**), dupa cum se va vedea mai jos:

```
fork: trei // desface firul in trei; cate unul pentru fiecare begin-end
begin
// codul pentru firul1
end
```

```
begin
// codul pentru firul2
end
begin
// codul pentru firul3
end
join // fuziunea firelor intr-un singur fir.
```

Fiecare instructiune intre **fork** si **join**, in acest caz cele trei blocuri begin-end, este executata concurent. Dupa terminarea celor trei fire, este executata urmatoarea instructiune dupa **join**. Trebuie sa existe certitudinea ca intre cele trei fire diferite nu exista interferente. De exemplu, nu se poate modifica continutul unui registru, in aceeasi perioada de ceas, in doua fire diferite.

Exemplu: Comanda semaforului de trafic.

Pentru a demonstra task-urile si evenimentele, se va prezenta un model de controlor de semafor de trafic.

```
// Model numeric pentru un controlul unui semafor de trafic.
// By Dan Hyde

module traffic;
parameter on = 1, off = 0, red_tics = 35,
amber_tics = 3, green_tics = 20;
reg clock, red, amber, green;
// va opri simularea dupa 1000 unitati de timp
initial begin: stop_at
#1000; $stop;
end
// initializeaza luminile si seteaza monitorizarea registrelor.
initial begin: Init
red = off; amber = off; green = off;
$display(" Timpul green amber red");
$monitor("%3d %b %b %b", $time, green, amber, red);
end
// task de asteptare pentru aparitiile fronturilor pozitive ale ceasului
// inainte de a stinge (off) luminile

task light;
output color;
input [31:0] tics;
begin
repeat(tics) // asteapta detectarea fronturilor pozitive ale ceasului
@(posedge clock);
color = off;
```

```
end
endtask
// forma de unda pentru o un ceas cu o perioada de doua unitati de timp
always begin: clock_wave
#1 clock = 0;
#1 clock = 1;
end
always begin: main_process
red = on;
light(red, red_tics); // cheama task-ul de asteptare
green = on;
light(green, green_tics);
amber = on;
light(amber, amber_tics);
end
endmodule
```

Iesirea simulatorului de semafor de trafic este urmatoarea:

```
Time green amber red
0 0 0 1
70 1 0 0
110 0 1 0
116 0 0 1
186 1 0 0
226 0 1 0
232 0 0 1
302 1 0 0
342 0 1 0
348 0 0 1
418 1 0 0
458 0 1 0
464 0 0 1
534 1 0 0
574 0 1 0
580 0 0 1
650 1 0 0
690 0 1 0
696 0 0 1
766 1 0 0
806 0 1 0
812 0 0 1
882 1 0 0
922 0 1 0
928 0 0 1
998 1 0 0
Stop at simulation time 1000
```

Bibliografie

1. Cadence Design Systems, Inc., *Verilog-XL Reference Manual.*

2. Open Verilog International (OVI), *Verilog HDL Language Reference Manual (LRM)*, 15466 Los Gatos Boulevard, Suite 109-071, Los Gatos, CA 95032; Tel: (408)353-8899, Fax: (408) 353-8869, Email: OVI@netcom.com, $100.

3. Sternheim, E. , R. Singh, Y. Trivedi, R. Madhaven and W. Stapleton, *Digital Design and Synthesis with Verilog HDL*, published by Automata Publishing Co., Cupertino, CA, 1993, ISBN 0-9627488-2-X, $65.

4. Thomas, Donald E., and Philip R. Moorby, *The Verilog Hardware Description Language*, second edition, published by Kluwer Academic Publishers, Norwell MA, 1994, ISBN 0-7923- 9523-9, $98, includes DOS version of VeriWell simulator and programs on diskette.

5. Bhasker, J., A Verilog HDL Primer, Star Galaxy Press, 1058 Treeline Drive, Allentown, PA18103, 1997, ISBN 0-9656277-4-8, $60.

6. Wellspring Solutions, Inc., *VeriWell User's Guide 1.2*, August, 1994, part of free distribution of VeriWell, available online.

7. World Wide Web Pages:

FAQ for comp.lang.verilog - http://www.comit.com/~rajesh/verilog/faq/alt_FAQ.html

comp.lang.verilog archives - http://www.siliconlogic.com/Verilog/

Cadence Design Systems, Inc. - http://www.cadence.com/

Exemple privind utilizarea limbajului Verilog pentru simularea sistemelor numerice.

```
// 1. Latch realizat cu porti NAND
module ffNand;
wire q, qBar;
reg preset, clear;
nand #1
g1 (q, qBar, preset),
g2 (qBar, q, clear);
initial
begin
// two slashes introduce a single line comment
$monitor ($time,,
"Preset = %b clear = %b q = %b qBar = %b",
preset, clear, q, qBar);
//waveform for simulating the nand flip flop
#10 preset = 0; clear = 1;
#10 preset = 1;
#10 clear = 0;
#10 clear = 1;
#10 $finish;
end
endmodule

// 2. Modul contor descris la nivel comportamental .

module m16Behav (value, clock, fifteen, altFifteen);
output [3:0] value;
reg [3:0] value;
output fifteen,
altFifteen;
reg fifteen,
altFifteen;
input clock;
initial
value = 0;
always
begin
@(negedge clock) #10 value = value + 1;
if (value == 15)
```

```
begin
altFifteen = 1;
fifteen = 1;
end
else
begin
altFifteen = 0;
fifteen = 0;
end
end
endmodule

//3. Modul contor descris la nivel structural

module m16 (value, clock, fifteen, altFifteen);
output [3:0] value;
output fifteen,
altFifteen;
input clock;
dEdgeFF a (value[0], clock, ~value[0]),
b (value[1], clock, value[1] ^ value[0]),
c (value[2], clock, value[2] ^ &value[1:0]),
d (value[3], clock, value[3] ^ &value[2:0]);
assign fifteen = value[0] & value[1] & value[2] &
value[3];
assign altFifteen = &value;
endmodule

// 4. Bistabil de tip D controlat pe front.

module dEdgeFF (q, clock, data);
output q;
reg q;
input clock, data;
initial
q = 0;
always
@(negedge clock) #10 q = data;
endmodule

//5. Ceas/orologiu pentru contor.

module m555 (clock);
output clock;
reg clock;
initial
#5 clock = 1;
always
#50 clock = ~ clock;
Endmodule

//6. Generator de semnal triunghiular.
```

```
module triangle(wave);
output[3:0] wave;
reg[3:0] wave;
reg clock;
reg[3:0] acc, inc;
initial begin
acc=0;
inc=0;
$display("Time clock wave");
$monitor("%3d %b %b",$time,clock,wave);
#110;$stop;
end
//forma de unda a ceasului
always
begin
#1 clock=0;
#1 clock=1;
end
//operarea triangle
always
begin
@(posedge clock);
if(wave == 15)
begin
inc = -1;
end
else if (wave == 0)
begin
inc = 1;
end
acc=acc+inc;
wave = acc;
end
endmodule

//7. Exemplu de simulare a contorului pe 4 biti.

/*Intr-un fisier verilog denumit board.v vor fi plasate
modulele: m555(clock); dEdgeFF (q, clock, data); m16 (value,
clock, fifteen, altFifteen);sau m16Behav (value, clock, fifteen,
altFifteen); */

//1. Modul contor descris la nivel comportamental .

/*module m16Behav (value, clock, fifteen, altFifteen);
output [3:0] value;
reg [3:0] value;
output fifteen,
altFifteen;
reg fifteen,
altFifteen;
```

```
input clock;
initial
value = 0;
always
begin
@(negedge clock) #10 value = value + 1;
if (value == 15)
begin
altFifteen = 1;
fifteen = 1;
end
else
begin
altFifteen = 0;
fifteen = 0;
end
end
endmodule
*/

//2. Modul contor descris la nivel structural.

module m16 (value, clock, fifteen, altFifteen);
output [3:0] value;
output fifteen,
altFifteen;
input clock;
dEdgeFF a (value[0], clock, ~value[0]),
b (value[1], clock, value[1] ^ value[0]),
c (value[2], clock, value[2] ^ &value[1:0]),
d (value[3], clock, value[3] ^ &value[2:0]);
assign fifteen = value[0] & value[1] & value[2] &
value[3];
assign altFifteen = &value;
endmodule
//3. Bistabil de tip D controlat pe front.
module dEdgeFF (q, clock, data);
output q;
reg q;
input clock, data;
initial
q = 0;
always
@(negedge clock) #10 q = data;
endmodule

//4. Ceas/orologiu pentru contor.

module m555 (clock);
output clock;
reg clock;
initial
```

```
#5 clock = 1;
always
#50 clock = ~ clock;
endmodule
//5. Top level module
module board;
wire [3:0] count;
wire clock, f, of;
initial begin: stop_at
#3200; $stop;
end
m16 counter (count,clock,f,of);
m555 clockGen(clock);
always @(posedge clock)
$display($time,,,"count=%d,f=%d,of=%d",count,f,of);
endmodule
```