

## Ce este Verilog?

În prezent există două limbaje de descriere a circuitelor de hardware, denumite generic prin acronimul *HDL* (de la *Hardware Description Language*), unul dintre ele denumit Verilog iar cel de-al doilea VHDL.

- **Verilog** provine din concatenarea cuvintelor din limba engleză “**Very**” și “**logic**” (cât se poate de logic). Limbajul a fost dezvoltat de către corporația americană **Gateway Design System Corporation** între anii **1983 – 1985** și are o sintaxă asemănătoare cu cea a limbajului C.
- **VHDL** provine din concatenarea literei “**V**” extrase din abrevierea *VHSIC* (*Very High Speed Integrated Circuits*) cu “**HDL**”.

Verilog și VHDL sunt considerate două limbaje diferite de descriere hardware, între care există însă multe asemănări.

## De ce Verilog?

Verilog este un limbaj standardizat, cu numărul de standard IEEE-1364, care definește o colecție de rutine software cunoscută sub denumirea de *PLI* (*Programming Language Interface*). Aceste rutine realizează interfațarea dintre Verilog și alte programe, de obicei C. Standardizarea a favorizat apariția ulterioară a unor companii care au creat simulatoare pentru circuitele descrise cu ajutorul limbajului Verilog. Un astfel de simulator este programul *ModelSim*. *ModelSim* este un program care acceptă pentru simulare atât descrieri realizate în Verilog cât și descrieri în VHDL.

## Introducere în Verilog

Verilog este un limbaj structural și procedural cu ajutorul căruia se pot construi structuri de blocuri, în care un bloc (blocul elementar) este denumit **modul**. Modulul poate descrie un proiect sau o parte de proiect.

Limbajul Verilog poate descrie un model sau o parte a unui model de hardware. Modelele descrise în Verilog pot fi descrieri **comportamentale** sau descrieri **structurale**. Limbajul permite descrierea componentelor modelului hardware precum și a conectorilor destinați interconectării componentelor sale.

Pentru înțelegerea structurii unui modul, vor fi oferite exemple de modelare a unor circuite logice cunoscute (precum sumatorul binar, multiplexorul, etc). Până la abordarea acestor exemple, este necesară trecerea în revistă a elementelor de limbaj Verilog folosite la construirea acestor module.

### I. Tipurile de date

Semnalele emise la ieșirile dispozitivelor digitale de comandă și aplicate intrărilor dispozitivelor comandate diferă între ele, motiv pentru care sunt asociate diferențiat unui anumit tip de date. Tipurile de date practicate sunt, în mare, de două categorii:

- **date de tip registru**. Ele sunt destinate să stocheze o valoare; sunt asociate semnalelor de la ieșirea unui bistabil de exemplu. Ele servesc la modelarea unui tip abstract de stocare similar registrului fizic.
- **date de tip net**. Acestea nu pot stoca o valoare; ele servesc doar ca suport al semnalelor de la ieșirea porților logice, semnale transmise porților comandate prin fire/conexiuni de legătură.

Tipul de date folosit pentru semnale (desemnate drept variabile) trebuie în general declarat explicit în partea de început a modulului ce conține descrierea funcțională a circuitului logic supus modelării.

Verilog consideră însă tipul de date declarat implicit, fără a mai fi necesară o declarație explicită, atunci când semnalul este utilizat la conexiunea unei construcții structurale de blocuri incluse în codul modulului. Declarațiile variabilelor de tipul **reg** sau **wire** neînsoțite de o dimensiune, sunt considerate de către program ca având implicit dimensiunea de 1 bit.

Iată două exemple de declarații de tipuri de date, așa cum apar ele în codul Verilog al unui modul:

```
reg q_iesire_bistabil ; // unde q_iesire_bistabil este numele dat unui semnal de tip reg de 1 bit
wire iesire_poarta_and ; // unde iesire_poarta_and este numele dat unui semnal tip wire de 1 bit
```

Cele două linii de cod Verilog de mai sus conțin după caracterul ";" un dublu slash "//" destinat așternerii unui comentariu. Verilog ignoră întotdeauna textul în linie de după caracterul "//", sau textul de pe mai multe linii, încadrat între "/\*" și "\*/".

### 1.1 Tipurile *net* de date

Se utilizează în descrierile structurale ale conexiunilor modelului descris. Cu excepția tipului particular **triereg**, celelalte tipuri **net** nu stochează valori. Ele primesc valoarea driverelor (dispozitivelor de comandă) care le comandă. Fiecare tip de **net** are o funcționalitate specifică modelării tipului respectiv de hardware.

Mai jos este prezentată lista tipurilor definite, conform Standardului IEEE *Language Reference Manual (LRM)*, pentru utilizare în Verilog:

Tipul net	Destinația
<b>wire , tri</b>	Conductor/fir simplu de conectare
<b>wor , trior</b>	Ieșiri OR cablate împreună
<b>wand, triand</b>	Ieșiri AND cablate împreună
<b>tri0</b>	Setare pe 0 logic în cazul existenței a 3 stări
<b>tri1</b>	Setare pe 1 logic în cazul existenței a 3 stări
<b>supply0</b>	Setează pe 0 logic intensitatea alimentării
<b>supply1</b>	Setează pe 1 logic intensitatea alimentării
<b>triereg</b>	Stochează ultima valoare în cazul apariției celei de-a 3-a stări (se referă la intensitatea capacității)

Dintre toate tipurile de **net**, tipul **wire** este cel mai des utilizat.

#### Reguli de știut:

- Dacă o rețea **net** este comandată de mai mulți driveri (de exemplu ieșirile a două porți legate împreună), atunci valoarea din rețea este aliniată tipului de rețea (**wire**, **wand**, **wor**, etc).
- Pentru un **net** de tipul **wire**, dacă toate driverele au aceeași valoare, atunci **wire**-ul poartă acea valoare.
- Dacă însă toate driverele cu excepția unuia au valoarea **z**, atunci **wire**-ul poartă valoarea **non z**.

- Dacă două sau mai multe drivere **non z** au diferite intensități de comandă, atunci **wire**-ul poartă semnalul de comandă cel mai puternic (the **stronger**).
- Dacă două drivere cu intensități de semnal egale emit valori logice diferite, atunci **wire**-ul poartă valoarea logică **x** (necunoscută).
- O rețea (net) de tipul **triereg** funcționează asemenea uneia de tip **wire**, cu excepția că atunci când driverul rețelei trece în starea **z** (înalță impedanță), atunci rețeaua reține ultima valoare emisă de driver. Net-urile de tipul **triereg** sunt folosite pentru modelarea rețelelor capacitive.
- Tipurile **wand** sau **triand** de net operează ca un AND-cablat iar net-urile **wor** sau **trior** operează ca un OR-cablat.
- Tipurile **tri0** și **tri1** modelează net-uri în montaj *pull-down* și respectiv *pull-up*, prin rezistor. Când un **tri0** nu este comandat (nu are semnal), atunci valoarea sa este pe 0. Când un **tri1** nu este comandat, atunci valoarea sa este pe 1.
- Tipurile **supply0** și **supply1** servesc pentru modelarea rețelelelor conectate la masă, respectiv la plusul sursei (+Vcc / +Vdd) de alimentare.

Tipul **net** de date trebuie asociat unui semnal în toate cazurile când:

- semnalul se obține la ieșirea unui dispozitiv;
- variabila semnalului este declarată port de intrare (*input*) sau de intrare-ieșire (*inout*);
- variabila semnalului se află în membrul stâng al expresiei unei instrucțiuni de atribuire continuă (cuvânt-cheie *assign*).

### Exemple de utilizări de date, tipul net:

```
wire [7:0] Data; //Comentariu: variabila vector Data de 8 biți, tipul wire
triereg (large) C1;
wire Q = A || B; //var. Q, tip wire, asigneata printr-o atribuire continua implicita
wire [7:0] Array [0:255][0:255][0:255]; /* tablou multidimensional din 3 module de
256 vectori a 8 biți. Comentariu pe 2 linii de cod sursa*/
```

Notă:

- **supply1** și **supply0** se folosesc doar pentru declararea sursei rețelei și respectiv a masei rețelei.

### 1.2 Tipurile *register* de date

Iată lista tipurilor register de date, definite conform LRM, pentru utilizare în Verilog:

Date tip register	Destinația
<b>reg</b>	destinat variabilelor fără semn, cu orice număr de biți
<b>integer</b>	alocă variabilei 32 de biți la stocare, cu semn sau fără semn
<b>time</b>	alocă variabilei 64 de biți, fără semn
<b>real , realtime</b>	alocă variabilei 32 de biți la stocare, destinate reprezentării variabilelor în virgulă flotantă simplă precizie

- **reg** este destinat modelării dispozitivelor digitale secvențiale ce stochează date;
- **reg** stochează valori logice și constante numerice. Nu stochează valori de intensități;
- **reg** este utilizat numai în blocurile de instrucțiuni procedurale, unde utilizarea acestui tip de date este obligatorie;

- datele semnalelor situate în membrul stâng al instrucțiunilor procedurale de atribuire trebuie să fie întotdeauna de tipul **reg**.
- blocurile procedurale încep întotdeauna cu cuvântul-cheie **initial** sau **always**.
- **reg** reține valoarea stocată până când o altă instrucțiune de atribuire suprascrive acea valoare;
- se pot crea tablouri de variabile de tip **reg** denumite memorii;

Setul de valori logice cu care pot fi atribuite variabilele de tipul **reg** și **wire** sunt **0**, **1**, **X** și **Z**. Valoarea **X** reprezintă o valoare necunoscută iar **Z** este înalta impedanță proprie porților TSL (Three State Logic). Valoarea **0** corespunde stării logice **false** iar **1** corespunde stării logice **true**.

Verilog efectuează întotdeauna inițializarea variabilelor/semnalelor în momentul declarării lor.

Valoarea logică a variabilelor de tipul **wire** și **reg** este inițializată cu **X** la declararea tipului de date și ea devine efectivă la momentul timp zero al simulării (adică la startul procesului de simulare) chiar dacă, pentru tipul **reg** există prevăzută o atribuire de valoare printr-un bloc procedural de tip **initial**.

Valoarea logică a variabilelor de tipul **wire** de pe firele neconectate este atribuită în mod implicit cu valoarea **Z** de către programul de simulare.

### Exemple de declarații de date tip register (reg, integer, time):

```
reg [7:0] Data; // variabila vector Data exprimata pe 8 biti
integer Int; // variabila Int exprimata pe 32 de biti, cu sau fara semn
time Timpul; // variabila Timpul, exprimata pe 64 de biti, fara semn
reg [15:0] Memory [0:1023]; // Memory, o memorie cu 1024 locatii, a 16 biti locatia
reg [11:0] A = 8'd511; // variabilei A de 12 biti i se atribuie constanta 'd255
reg [7:0] Array [0:255][0:255][0:255]; /* tablou multidimensional de vectori
de 8 biti.*/
```

#### Notă:

- **reg** sunt utilizate pentru descrieri/modelări de circuite logice secvențiale;
- **integer** sunt utilizate pentru variabile ciclate și la calcule;
- **real** sunt utilizate în modulele sistem;
- **time** și **realtime** sunt utilizate la stocarea timpilor de simulare din modulele *testbench* generatoare de stimuli. Această stocare este utilizată ulterior la afișări, în task-urile *\$display* și *\$monitor*.

Variabilelor de tipul **reg** li se pot atribui constante numerice dar doar ca numere întregi și fără semn. Variabilelor de tipul **integer** și de tipul **real** li se pot atribui constante numerice care pot fi numere întregi și respectiv numere reale, cu sau fără semn.

#### Exemplu:

```
reg [7:0] n = 8'h3C; // i se atribuie lui n un număr în format hexagesimal (adică 60 zecimal)
```

### I.2.1 Constantele numerice

Pentru exprimarea constantelor numerice se folosește următoarea sintaxă:

**dimensiune** **'baza\_de\_numerație** **valoare**

unde pentru **baza\_de\_numerație** se utilizează literele:

- a. **b** sau **B** (de la **binary**), care atestă că **valoare** este exprimată în baza de numerație 2;
- b. **o** sau **O** (de la **octal**), care atestă că **valoare** este exprimată în baza de numerație 8 ;
- c. **d** sau **D** (de la **decimal**), care atestă că **valoare** este exprimată în baza de numerație 10 ;
- d. **h** sau **H** (de la **hexadecimal**), care atestă că **valoare** este exprimată în baza de numerație 16.

Notă: la pagina 11 sunt prezentate pe larg exemple de modelare și simulare constante numerice.

În sintaxă, **dimensiune** este un număr zecimal care fixează numărul biților echivalentului binar al numărului exprimat de **valoare**. Acești biți dau valoarea finală a numărului binar stocat în calculator.

Când **dimensiune** este mai mic decât numărul biților formei binare echivalente lui **valoare**, atunci numărul acestor biți este trunchiat la numărul exprimat de **dimensiune** iar valoarea memorată în calculator va fi cea obținută în urma trunchierii (de la bitul cel mai din dreapta până la trunchiere).

Când **dimensiune** este mai mare decât numărul biților formei binare echivalente lui **valoare**, atunci biții formei binare echivalente se păstrează și se completează la stânga, cât prevede **dimensiune**, astfel:

- cu 0 (zero), dacă bitul cel mai din stânga al numărului binar este 0 sau 1;
- cu X sau cu Z, dacă bitul cel mai din stânga al numărului binar este X sau respectiv Z;

Fie declarațiile a 6 variabile tipul reg, de 12 biți de mai jos și atribuirile de constante numerice prezentate în tabel. Din analizarea acestora, se poate constata cum acționează regulile reprezentării.

De exemplu, **reg [11:0] a,b,c,d,e,f; // este declararea variabilor vector a,b,c,d,e,f de 12 biți**

Numele variabilei	Valoarea atribuită	Numarul binar stocat (pe 12 biți)	Correspondentul în hexagesimal
a	6'b <b>0X1</b>	0000_0000_00 <b>X1</b>	00X
b	6'b <b>1Z1</b>	0000_0000_01 <b>Z1</b>	00Z
c	6'b <b>X1</b>	0000_00XX_XX <b>X1</b>	0XX
d	6'b <b>Z01</b>	0000_00ZZ_ZZ <b>01</b>	0ZZ
e	3'b <b>0111X</b>	0000_0000_011 <b>X</b>	00X
f	3'b <b>01X11</b>	0000_0000_0 <b>X11</b>	00X

Observație: în reprezentarea în binar, fiecărei cifre hexagesimale i se rezervă 4 biți. De exemplu:

- lui F hexagesimal îi corespunde în binar secvența 1111 (adică 15 zecimal);
- lui FF hexagesimal îi corespunde în binar secvența 1111\_1111 (adică 255 zecimal);
- lui FFF hexagesimal îi corespunde în binar secvența 1111\_1111\_1111 (adică 4095 zecimal);
- lui AF hexagesimal îi corespunde în binar secvența 1010\_1111 (adică 175 zecimal);
- lui FA hexagesimal îi corespunde în binar secvența 1111\_1010 (adică 250 zecimal).

În tabelul următor sunt prezentate exemple de numere atribuite unor variabile de tipul **integer** și forma lor binară sub care vor fi stocate în computer (vezi și exemplele din Anexa 1, la pagina 15).

Numărul citat în instrucț. de atribuire	Numărul evaluat de program, memorat și afișat cu task-urile \$display și \$monitor (numerele sunt considerate constante numerice de tipul integer)
<b>11</b>	Numărul zecimal unsprezece. Declarat așa, programul îl consideră implicit drept număr zecimal. În computer este stocată secvența binară 0000_0000_0000_0000_0000_0000_0000_1011.
<b>-11</b>	Numărul zecimal negativ -11. În computer este stocată secvența codului complementar al lui -11 adică 1111_1111_1111_1111_1111_1111_1111_0101.
<b>4'b1011</b>	Număr binar evaluat pe 4 biți (corespunzător lui 11 zecimal) și stocat pe 32 de biți (v. mai sus).
<b>3'b1011</b>	Număr binar (echivalent cu 11 zecimal), dar evaluat pentru primii 3 biți, deci 011 binar (3 zecimal) stocat sub forma 0000_0000_0000_0000_0000_0000_0000_0011.
<b>-2'b0111</b>	Numărul binar negativ -0111 (corespunzător lui -7 zecimal), este evaluat pentru primii 2 biți, devenind -11 binar (-3 zec.). În computer este reținut codul complementar al lui -3, pe 32 de biți, adică: 1111_1111_1111_1111_1111_1111_1111_1101.
<b>-8'd3</b>	Numărul zecimal -3, evaluat în binar pe 8 biți, conduce la -0000_0011. Restul biților până la 32 sunt completați de program cu 0. Evaluarea în computer se face în binar, în cod complementar, adică 1111_1111_1111_1111_1111_1111_1111_1101 (doar biții de mărime).

<b>'b1101</b>	Numărul 13 binar. Evaluarea în computer se va realiza pe 32 biți, adică prin secvența: 0000_0000_0000_0000_0000_0000_0000_1101 proprie tipului de date <b>integer</b> .
<b>8'b111</b>	Numărul 7 binar. Evaluarea în computer se va realiza pe 8 biți (adică secvența 0000_0111) care se completează cu cifre de 0 până la dimensiunea de 32 biți proprie tipului de date <b>integer</b> .
<b>8'b0000_0111</b>	Același număr 7 în binar pe 8 biți, reluat, în care s-a introdus caracterul separator " " cu scopul de a ușura citirea numerelor binare lungi (procedeu admis în Verilog).
<b>3'b0110_0111</b>	Numărul binar echivalent lui 103 zecimal trunchiat la valoarea primilor 3 biți din dreapta, adică la secvența 111 (corespunzător lui 7 zecimal). Secvența reținută în computer va fi: 0000_0000_0000_0000_0000_0000_0000_0111.
<b>'d511</b>	Numărul zecimal 511. Computerul va reține secvența: 0000_0000_0000_0000_0001_1111_1111.
<b>8'd511</b>	Același număr zecimal 511, trunchiat în binar la primii 8 biți adică la secvența 1111_1111, care în computer devine: 0000_0000_0000_0000_0000_0000_1111_1111 = 255 <sub>(10)</sub>

### I.2.2 Constante numerice, tipul real

- Verilog admite numerele reale, sub formă de constante numerice sau atribuite variabilelor.
- Verilog convertește numerele reale în numere întregi prin rotunjire la cea mai apropiată cifră întreagă, atunci când sunt convertite în binar în vederea procesării lor în calculator.
- Tipul numerelor reale nu admite pe Z sau pe X.
- Numerele reale pot fi specificate fie în zecimal, fie în notație științifică, adică:  
 prin **< valoare > . < valoare >**, de exemplu: 314,15, -0.556, 13.568.  
 sau prin **< mantisă > E < exponent >**, de exemplu: 3.1415E2, -5,56E-1, 1.3568E1.

#### Exemplu de cod Verilog:

```
real U=12.625;
$display("U=%d, =%b", U, U); // pentru afisare valoare U in zecimal si binar
Rezultat: 13 (zecimal) și 0000_0000_0000_0000_0000_0000_1101 (13 binar pe 32 biți).
```

### I.2.3 Datele de tip string

O secvență de caractere, încadrată între ghilimele, formează un șir. Fiecare literă/character din șir este reprezentat pe 8 biți și este tratat drept un întreg pozitiv.

În exemplul de mai jos, stocarea literelor **RUN** necesită 3 cuvinte binare a 8 biți cuvântul, deci 24 biți.

#### Exemplu:

```
parameter numberChar = 3; // numberChar este declarat parametru, egal cu 3
reg[numberChar*8-1:0] message; /*semnalul denumit message este declarat tip reg
                                cu dimensiunea [23:0], adica de 24 de biti*/
message <= „RUN”; // variabilei message i se atribuie procedural sirul RUN.
```

## II. Operatorii

Servesc la modelarea în Verilog a expresiilor funcțiilor logice. Notă: exemple de module cu operatori scrise în Verilog și simularea acestora în ModelSim se găsesc la pagina 13.

### II.1 Operatorii aritmetici (Arithmetic Operators)

- “ + ”** adunare (unar și binar);  $a = 4'b1010$ ;  $\rightarrow a + 1 = 1011$
- “ - ”** scădere (unar și binar);  $b = 4'b1111$ ;  $c = 4'b1001$ ;  $\rightarrow b - c = 0110$ .
- “ \* ”** înmulțire;  $3 * 2 = 6$
- “ / ”** împărțire;  $8 / 2 = 4$
- “ % ”** modulo;  $8 \% 3 = 2$  (întoarce câtul întreg al împărțirii)

### II.2 Operatorii relaționali (Relational Operators)

Sunt utilizați pentru compararea între ei a doi operanzi sau a două expresii.



Rezultatul operației conduce la o valoare logică, valoarea 1 (pentru “true” = adevărat) sau valoarea 0 (pentru “false” = fals). Simbolurile utilizate pentru acești operatori sunt

- “ > ” semnifică: **mai mare**;
- “ < ” semnifică: **mai mic**;
- “ >= ” semnifică: **mai mare sau egal**;
- “ <= ” semnifică: **mai mic sau egal**.

### II.3 Operatorii de egalitate (*Equality Operators*)

- “ == ” operatorul de egalitate logică (*Logical Equality operator*);
- “ != ” operatorul de inegalitate logică (*Logical Inequality operator*);
- “ === ” operatorul de egalitate cu selecție (*Case Equality operator*);
- “ !== ” operatorul de inegalitate cu selecție (*Case Inequality operator*).

Acești operatori sunt folosiți pentru compararea bit cu bit a doi operanzi. Rezultatul operației conduce la o valoare logică, anume la 1 (“True”) în caz că egalitatea/inegalitatea se confirmă, sau la 0 (“False”) în caz că egalitatea/inegalitatea nu se confirmă. Dacă unul dintre operanzi este mai scurt, se aduce la lungimea celui mai lung prin completare la stânga cu zero-uri.

- Pentru operatorii == și != rezultatul este X, dacă unul dintre operanzi conține un X sau un Z
- Pentru operatorii === și !== biții X și Z sunt acceptați în comparație, dar trebuie să se potrivească și respectiv să nu se potrivească pentru ca rezultatul să fie 1 (true).

### II.4 Operatorii logici (*Logical Operators*)

- “ && ” este operatorul **AND logic**;
- “ || ” este operatorul **OR logic**;
- “ ! ” este operatorul **NOT logic** sau negație unară (a unei variabile de 1 bit).

Operatorii logici operează asupra operanzilor logici și returnează o valoare logică, de exemplu 1 (true) sau 0 (false). Sunt utilizați în mod specific în instrucțiunile **if** și **while**. Nu trebuie confundați cu operatorii booleani bitwise. De exemplu, “!” este operatorul logic **NOT** iar “~” este operatorul **bitwise NOT**. Primul neagă, de exemplu !(2==3) din care rezultă 1 (true). Al doilea complementează biții, de exemplu ~ {1,1,0,1} din care rezultă 0010 (aici acoladele au concatenat cei 4 biți dând 1101).

Rezultatul acțiunii operatorului este o valoare logică (un scalar) și anume:

- 1, dacă expresia este adevărată („true”); de exemplu, 1 && 1 = 1, 1 || 0 = 1, !(1 && 0) = 1
- 0, dacă expresia este falsă („false”); de exemplu, 1 && 0 = 0, 0 || 0 = 0, !(1 && 1) = 0.

### II.5 Operatorii logici destinați operanzilor vectori (*Bit-wise Operators*)

- “ ~ ” = **NOT** adică complementare aplicată fiecărui bit al unui operand (*bitwise negation*);
- “ & ” = **AND** aplicat între biții de același rang ai doi operanzi (*bi-twise AND*);
- “ | ” = **OR** aplicat între biții de același rang ai doi operanzi (*bit-wise OR*);
- “ ^ ” = **XOR** aplicat între biții de același rang ai doi operanzi (*bit-wise XOR*);
- “ ~^ ” = **XNOR** aplicat între biții de același rang ai doi operanzi (*bit-wise XNOR*).

Operatorii logici pe vectori operează “bit cu bit” între biții de același rang ai doi operanzi. Rezultatul operației este o secvență de biți, de lungimea operandului cu cei mai mulți biți. Rezultatul este deci un vector. Acest vector este evaluat cu 0 dacă toți biții săi sunt 0 sau este evaluat cu 1 dacă toți biții săi sunt 1.

### II.6 Operatorii de reducere (*Reduction Operators*)

Operatorii de reducere se aplică unui singur operand vector și conduc la un rezultat de un singur bit.

- “ & ” realizează funcția **AND** între toți biții unui operand vector; de exemplu: &1011 = 0
- “ ~& ” inversează rezultatul obținut după efectuare **AND** între toți biții vectorului; ~&1110 = 1
- “ | ” realizează funcția **OR** între toți biții operandului vector; de exemplu: | 1000 = 1
- “ ~| ” inversează rezultatul după efectuare **OR** între toți biții operandului vector; ~| 0001 = 0
- “ ^ ” realizează funcția **XOR** între toți biții operandului vector; de exemplu: ^ 1001 = 0
- “ ~^ ” inversează rezultatul după efectuare **XOR** între biții operandului vector; ~^ 1100 = 1

## II.7 Operatorii de deplasare (*Shift Operators*)

- “ >> n ” realizează deplasarea unei secvențe de biți spre dreapta, cu n pași (1110 >> 2=0011);
- “ << n ” realizează deplasarea unei secvențe de biți spre stânga, cu n pași (1110 << 2=1000).

Operatorul se aplică unui singur operand și determină deplasarea cu **n** poziții a biților operandului, plasat în stânga operatorului, în direcția precizată de săgețile operatorului. Pozițiile părăsite devin 0.

## II.8 Operatorii condiționali (*Conditional Operators*)

<condiție> ? <expresie1> : <expresie0>

Dacă expresia <condiție> este adevărată, operatorul întoarce valoarea lui <expresie1>, altfel operatorul întoarce valoarea lui <expresie0>.

## II.9 Operatorul de concatenare (*Concatenation Operator*)

Operatorul utilizează acolade, între care se succed, separate prin virgulă, caracterele ce urmează a fi concatenate (grupate împreună).

De exemplu, fie variabilele a,b,c:

a=1111\_0000; b=1001; c=1010\_1010, caz în care concatenarea de mai jos conduce la  
{a, b, c, 4'b1100} = 1111\_0000\_1001\_1010\_1010\_1100

Se precizează că sunt admise în concatenare doar numere a căror dimensiune este declarată în prealabil.

## II.10 Operatorul de replicare (*Replication Operator*)

Operatorul este utilizat la replicarea (repetarea) de un număr n de ori a unui grup de biți. De exemplu, să presupunem o variabilă vector de 4 biți, egală cu numărul binar 4'b1011, supusă replicării de 4 ori:

{4{4'b1011}} ==> 1011\_1011\_1011\_1011 // la simulare in ModelSim

sau

{5{x}} = {x,x,x,x,x} ==> xxxxx, // aici se concateneaza 5 caractere x.

## III. Instrucțiuni (*Statements*)

La modelările în limbaj Verilog se utilizează instrucțiuni de atribuire concurente și instrucțiuni de atribuire secvențiale. Notă: la pagina 20 sunt prezentate exemple de instrucțiuni simulate în ModelSim.

### III.1 Instrucțiunile de atribuire concurente (*Concurrent/Parallel Statements*)

Instrucțiunile utilizate sunt **assign**, **initial**, **always**. Ele se execută concomitent, în paralel.

- Instrucțiunea **assign** desemnează atribuiri continue; cuvântul assign precede o singură atribuire.
- Instrucțiunea **initial** desemnează un corp de atribuiri procedurale care se execută o singură dată. Atribuirile corpului **initial** pot fi reluate doar la o reluare a procesului de simulare.



- Instrucțiunea **always** desemnează de asemenea un corp de atribuiri procedurale, care însă se execută continuu. Ciclează la infinit putând îngheța simularea sau, până la împlinirea unei condiții (o întârziere sau un control al unui eveniment) sau până la terminarea simulării la comanda unei funcții sau a unui task de sistem.

### III.2 Instrucțiunile de atribuire secvențială (*Sequential Statements*)

Instrucțiunile de atribuire secvențială apar doar în corpul instrucțiunilor concurente **initial** și **always** și se execută strict în succesiunea în care sunt listate în codul sursă.

Când după **initial** și **always** se înscriu două sau mai multe instrucțiuni secvențiale de atribuire, ele trebuie grupate într-un bloc încadrat de cuvintele cheie **begin** și **end**. Acest bloc se comportă ca o singură instrucțiune.

În limbajul Verilog există o distincție importantă între atribuirea continuă și atribuirea procedurală. Instrucțiunea de atribuire continuă **assign** este utilizată la modelarea logicii combinaționale. Instrucțiunile de atribuire procedurale **initial** și **always** sunt utilizate la modelarea logicii secvențiale.

Atribuirea **assign** atribuie valori variabilelor de tipul **wire** care sunt reevaluate/actualizate de fiecare dată când unul dintre operandii de intrare își modifică valoarea. În exemplul de mai jos, de fiecare dată când operandii **in1** și **in2** (oricare) își schimbă valoarea, variabila **out** este reevaluată:

**wire** out ; // este declarata variabila **out**, de tip **wire**

**assign** out = in1 & in2 ; // se efectueaza atribuirea continua a variabilei **out**

**Notă:** linia de cod *assign out* face inutilă declarația *wire out*, considerată implicită de către program

Atribuirile procedurale **initial** și **always** prezintă două forme de atribuire: **blocantă** și **neblocantă**.

Atribuirea blocantă (*blocking assignment*) folosește operatorul „**=**”. În cazul ei, o instrucțiune este trecută la execuție numai după ce instrucțiunea precedentă a blocului a fost executată.

Atribuirea neblocantă (*non-blocking assignment*) folosește operatorul „**<=**”. Ea evaluează termenul din dreapta operatorului și atribuie valoarea obținută termenului din stânga. Toate atribuirile sunt executate concomitent sau în ordinea dictată de un *#delay* dacă acesta există (care precede întotdeauna instrucțiunea de atribuire).

Să presupunem, în următoarele două exemple, că variabila **a** are inițial valoarea **a = 1**.

**- exemplu de modelare a unei atribuiri blocante**

**reg** a;

**always @ (posedge clk)** // pe frontul pozitiv de clock intra in functie always

**begin**

a=a+1; // aici are loc atribuirea a=1+1=2

a=a+2; // după execuția atribuirii precedente are loc atribuirea a=2+2=4

**end**

**- exemplu de modelare a unei atribuiri neblocante**

**reg** a;

**always @ (posedge clk)** // pe frontul pozitiv de clock intra in functie always

**begin**

a<=a+1; // aici are loc atribuirea a=1+1=2

a<=a+2; //simultan cu atribuirea precedentă are loc atribuirea a=1+2=3 (rezultatul final).

**end**

După cum se poate observa, cele două atribuiri neblocaute utilizează aceeași valoare inițială a=1 la evaluarea expresiei membrului drept.

În cele două exemple de cod Verilog, după simbolul @ urmează între paranteze așa numita *sensitive list* care indică faptul că blocul **always** va fi declanșat (*triggered*) la producerea evenimentelor înscrise în listă (în cazul de mai sus, la producerea unui front crescător al semnalului de clock).

O observație importantă este că blocurile *always* nu pot atribui valori (nu pot comanda/dirija) tipului de date *wire*, însă pot atribui valori tipurilor *reg* și *integer*.

Operatorul „=” este folosit pentru atribuiri atât în logica combinațională cât și în logica secvențială:

- în logica combinațională este prezent alături de instrucțiunea **assign** având rol de atribuire continuă;
- în logica secvențială este prezent alături de instrucțiunile **initial** și **always** și este utilizat la atribuiri secvențiale în interiorul blocurilor *begin-end*.

Un bloc procedural **always** poate exista fără lista *sensitive list*. În acest caz este nevoie de o întârziere (*delay*) pentru a evita ciclarea la infinit a instrucțiunii (ciclare care conduce la blocarea simulării). O astfel de întârziere (#5, ca cea din exemplul de mai jos) se plasează în fața instrucțiunii de atribuire. Acest *delay* întârzie execuția atribuirii cu 5 unități curente de timp. Unitatea de timp (ns, ps, etc) se setează de către utilizator (la alegere) înainte de procedura de simulare. De exemplu,

```
always
begin
    #5  clk = ~clk; // tot la 5 unitati de timp semnalul clk schimba de stare
end
```

De reținut:

în cazul atribuirilor **assign**, **initial**, și **always** membrul drept al atribuirii poate fi fie de tipul **wire**, fie de tipul **reg**.

- în cazul atribuirilor **assign**

Membrul stâng este întotdeauna un **wire**.

Deci:

```
assign wire = wire, reg
```

Exemplu:

```
reg A; //declarare tip reg de date pentru A
wire B, C; //declarare tip wire de date pentru B,C
```

```
assign C = A ^ B; // cand se modifica A si/sau B
assign C = B; // cand se modifica B
```

- în cazul blocurilor **always**, **initial**

Membrul stâng este întotdeauna un **reg**.

Deci:

```
always @ (.....)
begin
    reg = wire, reg
end
```

Exemplu:

```
wire A; //declarare tip wire de date pentru A
reg B, C, D; //declarare tip reg pentru A,B,C
```

```
always @ (A, B) // cand se modifica A si/sau B
begin
    C = A ^ B; // cand se modifica A si/sau B
    D = A; // cand se modifica A
end
```

**Exemple de numere întregi** (example of Integer Numbers)

Tipul Integer	Stocat în calculator (Stored as)
1	0000_0000_0000_0000_0000_0000_0001 (1 zecimal reprezentat în binar)
8'hAA	0000_0000_0000_0000_0000_0000_1010_1010
6'b10_0011	0000_0000_0000_0000_0000_0000_0010_0011
'hF	0000_0000_0000_0000_0000_0000_0000_1111
6'hCA	0000_0000_0000_0000_0000_0000_0000_1010)
6'hA	0000_0000_0000_0000_0000_0000_0000_1010)
16'bZ	ZZZZ_ZZZZ_ZZZZ_ZZZZ
8'bx	xxxx_xxxx

**Observație:**

În calculator, pentru tipul de date (numere) de tipul **integer** sunt rezervați 32 de biți. La stocare în calculator, numărul biților numărului binar dat este trunchiat, de la dreapta la stânga, la numărul de biți egal cu numărul înscris în stânga apostrofului; acest număr, după trunchiere, se va stoca pe 32 de biți.

**Explicații la exemplele de mai sus din tabel**

- **8'hAA** va fi stocat și afișat sub forma:

- în binar : **0000\_0000\_0000\_0000\_0000\_0000\_1010\_1010**
- în hexazecimal : **0000\_00AA**, (fiecărei tetrade binare îi corespunde o cifră hexagesimală)

- **6'b10\_0011** va fi stocat și afișat sub forma:

- în binar : **0000\_0000\_0000\_0000\_0000\_0000\_0010\_0011**
- în hexazecimal: **0000\_0023**

- **'hF** va fi stocat și afișat sub forma:

- în binar : **0000\_0000\_0000\_0000\_0000\_0000\_0000\_1111**
- în hexazecimal : **0000\_000F**, (fiecărei tetrade binare îi corespunde o cifră hexagesimală)

- **6'hCA**

- CA hexazecimal convertit în binar este: **1100\_1010**
- în binar pe 32 de biți : **0000\_0000\_0000\_0000\_0000\_0000\_1100\_1010**
- în binar trunchiat la 6 biți: **00\_1010**
- în binar trunchiat la 6 biți, pe 32 de biți : **0000\_0000\_0000\_0000\_0000\_0000\_0000\_1010**
- în hexazecimal : **0000\_000A** (fiecărei tetrade binare îi corespunde o cifră hexagesimală)

- **6'hA**

- A hexazecimal convertit în binar este: **1010**
- în binar pe 32 de biți : **0000\_0000\_0000\_0000\_0000\_0000\_0000\_1010**
- în binar trunchiat la 6 biți: **00\_1010**
- în binar trunchiat la 6 biți, pe 32 de biți : **0000\_0000\_0000\_0000\_0000\_0000\_0000\_1010**
- în hexazecimal : **0000\_000A** (fiecărei tetrade binare îi corespunde o cifră hexagesimală)

- **16'bz** va fi stocat și afișat sub forma:

- în binar : **0000\_0000\_0000\_0000\_zzzz\_zzzz\_zzzz\_zzzz** (totodată și forma de stocare)

- în hexazecimal : 0000\_zzzz
- **8'bx** va fi stocat și afișat sub forma:
  - în binar : 0000\_0000\_0000\_0000\_0000\_0000\_xxxx\_xxxx
  - în hexazecimal : 0000\_00xx (fiecărei tetrade binare îi corespunde o cifră hexagesimală)

**Corpul modului, creat în Verilog, pentru afișarea cu ModelSim a numerelor prezentate mai sus**

```
module afis_numere ();
integer a,b,c,d,e,f,g;
initial
begin
    a = 8'hAA;
    b = 6'b10_0011;
    c = 'hF;
    d = 6'hCA;
    e = 6'hA;
    f = 16'bz;
    g = 8'bx;

    $display("a=8'hAA [%h]= %h, [%b]= %b", a);
    $display("b=6'b10_0011 [%h]= %h, [%b]= %b", b);
    $display("c='hF [%h]= %h, [%b]= %b", c);
    $display("d=6'hCA [%h]= %h, [%b]= %b", d);
    $display("e=6'hA [%h]= %h, [%b]= %b", e);
    $display("f=16'bz [%h]= %h, [%b]= %b", f);
    $display("g=8'bx [%h]= %h, [%b]= %b", g);
end
endmodule
```

Rezultatele afișate în panoul *Transcript* urmare simulării în ModelSim:

```
a=8'hAA      [%h]= 0000_00aa, [%b]= 0000_0000_0000_0000_0000_0000_1010_1010
b=6'b10_0011 [%h]= 00000023, [%b]= 0000_0000_0000_0000_0000_0000_0010_0011
c='hF        [%h]= 0000000f, [%b]= 0000_0000_0000_0000_0000_0000_0000_1111
d=6'hCA      [%h]= 0000000a, [%b]= 0000_0000_0000_0000_0000_0000_0000_1010
e=6'hA       [%h]= 0000000a, [%b]= 0000_0000_0000_0000_0000_0000_0000_1010
f=16'bz      [%h]= 0000zzzz, [%b]= 0000_0000_0000_0000_zzzz_zzzz_zzzz_zzzz
g=8'bx       [%h]= 000000xx, [%b]= 0000_0000_0000_0000_0000_0000_xxxx_xxxx
```

## OPERATORII LIMBAJULUI VERILOG

### 1. OPERATORII DE EGALITATE – *Equality Operators*

Există două tipuri de operatori de egalitate și anume:

- operatori de egalitate/inegalitate **cu selecție** (*Case Equality / Case Inequality*);
- operatori de egalitate/inegalitate **logică** (*Logical Equality*).

Operatori	Descriere
<code>a === b</code>	Când a este egal cu b, putând include pe x și/sau z (tipul de <i>Case equality</i> )
<code>a !== b</code>	Când a nu e egal cu b, putând include pe x și/sau z (tipul de <i>Case inequality</i> )
<code>a == b</code>	a este egal cu b, rezultat ce poate fi necunoscut (Logical Equality)
<code>a != b</code>	a nu este egal cu b, rezultat ce poate fi necunoscut (Logical Inequality)

#### Notă :

- Pentru operatorii `===` și `!==`, operandii sunt comparați bit cu bit. În prealabil, dacă cei doi operanzi nu au aceeași lungime, operandul mai scurt este completat cu 0. Biții X și Z sunt admiși în comparație însă pentru ca rezultatul să fie 1(true) ei trebuie să ocupe aceeași poziție în secvențele binare ale operandilor. Rezultatul comparației este un 0 (false) sau un 1 (true), după caz.

- Pentru operatorii `==` și `!=` rezultatul este X dacă unul dintre operanzi conține un X sau un Z, altfel este un 0 (false) sau un 1 (true), după caz.

La simulare, ModelSim nu acceptă X sau Z în structura operandului decât dacă operandul este un număr binar. La operandii ne-binari compilarea este trecută cu bine, însă după setarea funcției **Start Simulation** este semnalată eroarea „Error loading design”. (vezi <D:\Home\Colocviu\Exercitiul 12>).

```
module equality_operators(); // vezi proiectul Proba1
```

```
//autor: svh, Mai 2010
```

```
initial
```

```
begin
```

```
// Case Equality
```

```
$display ("Case Equality");
```

```
$display ("  Expresii cu operator      Rezultat logic");
```

```
$display (" 4'bX001 === 4'bX001 ==>  %b", (4'bX001 === 4'bX001));
```

```
$display (" 4'bX0X1 === 4'bX001 ==>  %b", (4'bX0X1 === 4'bX001));
```

```
$display (" 4'bZ0X1 === 4'bZ0X1 ==>  %b", (4'bZ0X1 === 4'bZ0X1));
```

```
$display (" 4'bZ0X1 === 4'bZ001 ==>  %b", (4'bZ0X1 === 4'bZ001));
```

```
$display ();
```

```
// Case Inequality
```

```
$display ("Case Inequality");
```

```
$display ("  Expresii cu operator      Rezultat logic");
```

```
$display (" 4'bX0X1 !== 4'bX0X1 ==>  %b", (4'bX0X1 !== 4'bX0X1));
```

```

$display (" 4'bX0X1 !== 4'bX001 ==> %b", (4'bx0x1 !== 4'bx001));
$display (" 4'bZ0X1 !== 4'bZ001 ==> %b", (4'bz0x1 !== 4'bz001));
$display ();

// Logical Equality
$display ("Logical Equality");
$display ("  Expresii cu operator      Rezultat logic");
$display ("      5 == 5                ==> %b", (5 == 5));
$display ("      5 == 10              ==> %b", (5 == 10));
$display ("      'd15 == 'hF          ==> %b", ('d15 == 'hf));
$display ("      2'b10 == 2'b10       ==> %b", (2'b10 == 2'b10));
$display ("      2'b1X == 2'b1X       ==> %b", (2'b1x == 2'b1x));
$display ("      2'bZ1 == 2'bZ1       ==> %b", (2'bz1 == 2'bz1));
$display ();

// Logical Inequality
$display ("Logical Inequality");
$display ("  Expresii cu operator      Rezultat logic");
$display ("      2'b10 != 2'b10        ==> %b", (2'b10 != 2'b10));
$display ("      2'b1X != 2'b1X        ==> %b", (2'b1x != 2'b1x));
$display ("      2'b1Z != 2'b1Z        ==> %b", (2'b1z != 2'b1z));
$display ("      2'b1Z != 2'b1X        ==> %b", (2'b1z != 2'b1x));
$display ("      2'b10 != 2'b11        ==> %b", (2'b10 != 2'b11));
$display ();
#1 $stop;
end
endmodule

```

## 2. OPERATORII BITWISE – Bit-wise Operators

Operatorii se aplică operanzilor de tip vector. Acționează între biții de același rang ai doi operanzi. Există operatori de tipurile Negație, AND, OR, XOR și XNOR. Dacă cei doi operanzi sunt de lungimi diferite, cel cu lungimea mai scurtă este extins la lungimea celui alt prin completare la stânga cu zerouri.

```

module bitwise_operators(); // vezi proiectul Proba2
initial
begin
    // Bit Wise Negation
    $display ();
    $display (" Bit Wise Negation");
    $display (" ~0001 = %b", (~4'b0001));
    $display (" ~x001 = %b", (~4'bx001));
    $display (" ~z001 = %b", (~4'bz001));

    // Bit Wise AND
    $display ();

```



```
$display (" Bit Wise AND");
$display (" 0001 & 1001 = %b", (4'b0001 & 4'b1001));
$display (" 1001 & x001 = %b", (4'b1001 & 4'bx001));
$display (" 1001 & z001 = %b", (4'b1001 & 4'bz001));
$display (" x001 & z001 = %b", (4'bx001 & 4'bz001));

// Bit Wise OR
$display ();
$display (" Bit Wise OR");
$display (" 0001 | 1001 = %b", (4'b0001 | 4'b1001));
$display (" 0001 | x001 = %b", (4'b0001 | 4'bx001));
$display (" 1001 | x001 = %b", (4'b1001 | 4'bx001));
$display (" 0001 | z001 = %b", (4'b0001 | 4'bz001));
$display (" 1001 | z001 = %b", (4'b1001 | 4'bz001));
$display (" x001 | z001 = %b", (4'bx001 | 4'bz001));

// Bit Wise XOR
$display ();
$display (" Bit Wise XOR");
$display (" 0001 ^ 1001 = %b", (4'b0001 ^ 4'b1001));
$display (" 0001 ^ x001 = %b", (4'b0001 ^ 4'bx001));
$display (" 0001 ^ z001 = %b", (4'b1001 ^ 4'bz001));
$display (" x001 ^ z001 = %b", (4'bx001 ^ 4'bz001));

// Bit Wise XNOR
$display ();
$display (" Bit Wise XNOR");
$display (" 0001 ~^ 1001 = %b", (4'b0001 ~^ 4'b1001));
$display (" 0001 ~^ x001 = %b", (4'b0001 ~^ 4'bx001));
$display (" 0001 ~^ z001 = %b", (4'b0001 ~^ 4'bz001));
$display (" x001 ~^ z001 = %b", (4'bx001 ~^ 4'bz001));
$display ();
#10 $stop;
end
endmodule
```

---

### 3. OPERATORII DE REDUCERE – Reduction Operators

Ei execută operația între biții unui singur operand-vector (denumiți din acest motiv operatori unari) și produc drept rezultat un singur bit.

Operatorii sunt **&**, **~&**, **|**, **~|**, **^**, **~^** (sau **^~**) denumiți **AND**, **NAND**, **OR**, **NOR**, **XOR** respectiv **XNOR**. Operatorii unari **NAND** și **NOR** operează asemenea operanzilor **AND** și respectiv **OR** cu ieșirea negată.

```
module reduction_operators(); // vezi proiectul Proba3
initial
begin
```

```
// Bit Wise AND reduction
$display ();
$display (" Bit Wise AND reduction");
$display (" & 4'b1001 = %b", (& 4'b1001));
$display (" & 4'bx111 = %b", (& 4'bx111));
$display (" & 4'bz111 = %b", (& 4'bz111));
$display (" & 4'bx011 = %b", (& 4'bx011));
$display (" & 4'bz011 = %b", (& 4'bz011));

// Bit Wise NAND reduction
$display ();
$display (" Bit Wise NAND reduction");
$display (" ~& 4'b1001 = %b", (~& 4'b1001));
$display (" ~& 4'bx001 = %b", (~& 4'bx001));
$display (" ~& 4'bz001 = %b", (~& 4'bz001));
$display (" ~& 4'bx111 = %b", (~& 4'bx111));
$display (" ~& 4'bz111 = %b", (~& 4'bz111));
$display (" ~& 4'bxz11 = %b", (~& 4'bxz11));

// Bit Wise OR reduction
$display ();
$display (" Bit Wise OR reduction");
$display (" | 4'b1001 = %b", (| 4'b1001));
$display (" | 4'bx000 = %b", (| 4'bx000));
$display (" | 4'bz000 = %b", (| 4'bz000));
$display (" | 4'bx100 = %b", (| 4'bx100));
$display (" | 4'bz100 = %b", (| 4'bz100));
$display (" | 4'bxz00 = %b", (| 4'bxz00));

// Bit Wise NOR reduction
$display ();
$display (" Bit Wise NOR reduction");
$display (" ~| 4'b1001 = %b", (~| 4'b1001));
$display (" ~| 4'bx001 = %b", (~| 4'bx001));
$display (" ~| 4'bz001 = %b", (~| 4'bz001));
$display (" ~| 4'bx000 = %b", (~| 4'bx000));
$display (" ~| 4'bz000 = %b", (~| 4'bz000));

// Bit Wise XOR reduction
$display ();
$display (" Bit Wise XOR reduction");
$display (" ^ 4'b1001 = %b", (^ 4'b1001));
$display (" ^ 4'bx001 = %b", (^ 4'bx001));
$display (" ^ 4'bz001 = %b", (^ 4'bz001));
$display (" ^ 4'bxz00 = %b", (^ 4'bxz00));
```

```
// Bit Wise XNOR
$display ();
$display (" Bit Wise XNOR reduction");
$display (" ~^ 4'b1001 = %b", (~^ 4'b1001));
$display (" ~^ 4'b1001 = %b", (~^ 4'b1001));
$display (" ~^ 4'b1001 = %b", (~^ 4'b1001));
$display ();
#1 $stop;
end
endmodule
```

---

#### 4. OPERATORUL CONDIȚIONAL – Conditional Operator

Are formatul:

*(expresie\_conditie) ? expresie\_1 : expresie\_0.*

Se evaluează *expresie\_conditie* și dacă aceasta este îndeplinită, se întoarce valoarea rezultată din evaluarea lui *expresie\_1*, altfel se întoarce valoarea lui *expresie\_0*

```
module conditional_operator(); // vezi proiectul Proba4
wire out;
reg enable,data;
// Tri state buffer
assign out = (enable) ? data : 1'bz; // daca <enable> este true, atunci out=data, altfel out=z
initial
begin
$display (""); // afiseaza un rand liber
$display ("time\t enable\t data\t out"); // caracterele \t comanda o tabulare
$monitor ("%0d\t %b\t %b\t %b", $time,enable,data,out); //$time este o functie sistem de timp
enable = 0;
data = 0;
data <= #1 1;
enable <= #2 1;
data <= #3 0;
#4 $stop; // opreste procedura de simulare la 4 u.t. de la start
end
endmodule
```

---

#### 5. OPERATORUL DE CONCATENARE – Concatenation Operator

```
module concatenation_operator();
reg [3:0] a, b, c;
reg [7:0] d;
initial
begin
#1;
a = 4'b1111;
b = 4'bzzzz;
```

```
c = {a,b};
d = {a,b};
// concatenation operator
$display (" \n Concatenation two binary numbers");
$display (" {4'b1000, 4'bx00z} = %b ", {4'b1000, 4'bx00z});
$display (" \n a = %b, b = %b, c={a,b}= %b, d={a,b}= %b \n", a, b, c, d);
$display ("Se observa ca, dupa concatenare, c contine doar bitii lui b.");
$display ();
#1 $stop;
end
endmodule
```

---

## 6. OPERATORUL DE REPLICARE – Replication Operator

```
module replication_operator();
initial
begin
    // Replication
    $display (" Replication");
    $display (" {3{1110}} = %b \n", {3{4'b1110}}); //replicare de 3 ori a numarului binar 1110 de 4 cifre
    // Concatenation and replication
    $display (" Concatenation and replication");
    $display (" {2{1001,z}} = %b \n", {2{4'b1001,1'bz}}); //concatenare 1001 cu z si replicare de 2 ori a lui 1001z
    #1 $stop;
end
endmodule
```

---

## 7. OPERATORII DE DEPLASARE – Shift Operators

Operandul din stânga operatorului este deplasat cu numărul de poziții de bit indicat de numărul plasat în dreapta operatorului.

Pozițiile de bit rămase vacante se completează cu zero.

```
module shift_operators();
reg [3:0] a, b, c;
initial
begin
    $monitor ("time=%g a= %b b= %b c= %b", $time, a, b, c);
    a <= 4'b0000;
    b <= 4'b0000;
    #10 a <= 4'b1x1z;
    #20 b <= a << 1;
    #5 c <= a << 1;
    // Left Shift
    $display ("Left shift");
end
endmodule
```

```
$display (" 4'b1001 << 1 = %b", (4'b1001 << 1));
$display (" 4'b10x1 << 1 = %b", (4'b10x1 << 1));
$display (" 4'b10z1 << 1 = %b", (4'b10z1 << 1));
// Right Shift
$display ("Right shift");
$display (" 4'b1001 >> 1 = %b", (4'b1001 >> 1));
$display (" 4'b10x1 >> 1 = %b", (4'b10x1 >> 1));
$display (" 4'b10z1 >> 1 = %b", (4'b10z1 >> 1));
#10 $stop;
end
endmodule
```

**Observație:** instrucțiunea de shift așezată:  $c = 1 \ll a$  nu funcționează !

---

## 8. OPERATORII LOGICI – Logical Operators

- Operatorii sunt **!**, **&&**, **||** denumiți *Negație logică*, *AND logic* și respectiv *OR logic*.
- Expresiile legate prin operatorii **&&** și **||** sunt evaluate de la stânga spre dreapta
- Evaluarea se oprește imediat ce rezultatul devine cunoscut
- Rezultatul este ca valoare un scalar, și anume:
  - 0 dacă rezultatul este fals (*false*)
  - 1 dacă rezultatul este adevărat (*true*)
  - X dacă oricare dintre operanzi conține biți X (de valoare necunoscută) sau Z

```
module operatori_logici; // vezi proiectul Proba8
//autor: svh, Mai 2010
initial
begin
// Operatorul logic AND
$display("Logical AND");
$display(" 0 && 1 = ", 1'b0 && 1'b1); // rezultat: 0
$display(" 0 && X = ", 1'b0 && 1'bX); // rezultat: 0
$display(" 0 && Z = ", 1'b0 && 1'bZ); // rezultat: 0
$display(" 1 && 1 = ", 1'b1 && 1'b1); // rezultat: 1
$display(" 1 && X = ", 1'b1 && 1'bX); // rezultat: x
$display(" 1 && Z = ", 1'b1 && 1'bZ); // rezultat: x
$display(" X && X = ", 1'bX && 1'bX); // rezultat: x
$display(" X && Z = ", 1'bX && 1'bZ); // rezultat: x
$display();

// Operatorul logic OR
$display("Logical OR");
$display(" 0 || 1 = ", 1'b0 || 1'b1); // rezultat: 1
$display(" 0 || X = ", 1'b0 || 1'bX); // rezultat: x
$display(" 0 || Z = ", 1'b0 || 1'bZ); // rezultat: x
$display(" 1 || 1 = ", 1'b1 || 1'b1); // rezultat: 1
$display(" 1 || X = ", 1'b1 || 1'bX); // rezultat: 1
```

```
$display(" 1 || Z = ", 1'b1 || 1'bZ); // rezultat: 1
$display(" X || Z = ", 1'bX || 1'bZ); // rezultat: x
$display(" Z || Z = ", 1'bZ || 1'bZ); // rezultat: x
$display();

// Operatorul logic NOT (!)
$display("Logical NOT (!)");
$display(" ! 0 = ", ! 1'b0); // rezultat: 1
$display(" ! 1 = ", ! 1'b1); // rezultat: 0
$display(" ! X = ", ! 1'bX); // rezultat: x
$display(" ! Z = ", ! 1'bZ); // rezultat: x
$display();

// Operatorul logic NOT (~)
$display("Logical NOT (~)");
$display(" ~ 0 = ", ! 1'b0); // rezultat: 1
$display(" ~ 1 = ", ! 1'b1); // rezultat: 0
$display(" ~ X = ", ! 1'bX); // rezultat: x
$display(" ~ Z = ", ! 1'bZ); // rezultat: x

end
endmodule
```

## Atribuire secvențiale blocante și neblocante. Exemple de modelare și simulare

Atribuirea **blocantă** sau **blocking assignment**, materializată de operatorul “ = ”, indică execuția unei instrucțiuni de atribuire ce așteaptă terminarea instrucțiunii precedente, adică instrucțiunile se execută secvențial. Atribuirea blocantă folosește în logica combinațională următoarea sintaxă:

**variabila = expresie;**

Atribuirile **neblocante** sau **nonblocking assignment**, materializate de operatorul “ <= ” sunt executate în paralel. Atribuirile secvențiale neblocante sunt folosite la modelarea bistabilelor, registrelor, etc. cu sintaxa:

**variabila <= expresie;**

**Atenție! Verilog nu admite utilizarea celor două tipuri de atribuire în aceeași procedură.**

Pentru o mai bună înțelegere a celor două tipuri de atribuire se prezintă mai jos un exemplu cu 4 variabile (semnale) **a, b, c, d** cărora li se atribuie valori folosind tipul *blocking* de instrucțiuni de atribuire și cu alte 4 variabile **e, f, g, h** pentru tipul *nonblocking*.

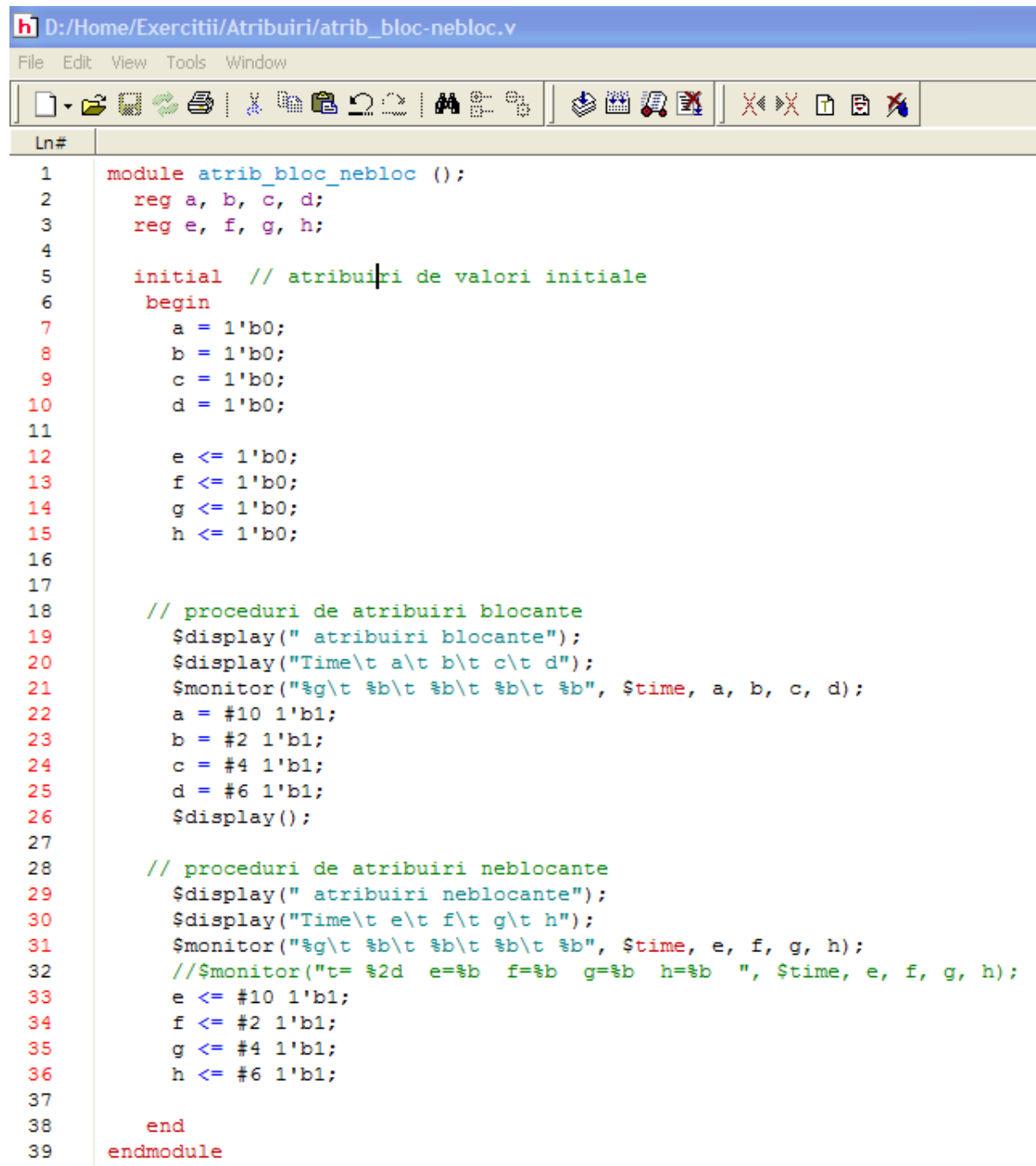
Task-ul sistem **\$display**, introdus în structura modulului, afișează în panoul **Transcript** al interfeței grafice *Model Sim* valoarea timpului curent de simulare cu ajutorul funcției sistem **\$time**, în format decimal, la momentul respectiv al procesului de simulare (v. mai jos fragmentul pastat intitulat Transcript, decupat cu *Paint* din fereastra *ModelSim*).

În imagini, alături de panoul **Transcript**, este prezentat panoul **list** (decupat din fereastra *ModelSim*), ce conține valorile atribuite variabilelor, afișate în dreptul întârzierilor (în ps) consemnate în codul Verilog



al modulului.

De asemenea, sunt prezentate valorile setate pentru unitatea de timp (1 ps) folosită de simulator, precum și pentru intervalul de timp ales pentru rulajul simulării (100 ps).



```

D:/Home/Exercitii/Atribuire/atrib_bloc_nebloc.v
File Edit View Tools Window
Ln#
1  module atrib_bloc_nebloc ();
2      reg a, b, c, d;
3      reg e, f, g, h;
4
5      initial // atribuire de valori initiale
6      begin
7          a = 1'b0;
8          b = 1'b0;
9          c = 1'b0;
10         d = 1'b0;
11
12         e <= 1'b0;
13         f <= 1'b0;
14         g <= 1'b0;
15         h <= 1'b0;
16
17
18         // proceduri de atribuire blocante
19         $display(" atribuire blocante");
20         $display("Time\t a\t b\t c\t d");
21         $monitor("%g\t %b\t %b\t %b\t %b", $time, a, b, c, d);
22         a = #10 1'b1;
23         b = #2 1'b1;
24         c = #4 1'b1;
25         d = #6 1'b1;
26         $display();
27
28         // proceduri de atribuire nebloccante
29         $display(" atribuire nebloccante");
30         $display("Time\t e\t f\t g\t h");
31         $monitor("%g\t %b\t %b\t %b\t %b", $time, e, f, g, h);
32         // $monitor("t= %2d e=%b f=%b g=%b h=%b ", $time, e, f, g, h);
33         e <= #10 1'b1;
34         f <= #2 1'b1;
35         g <= #4 1'b1;
36         h <= #6 1'b1;
37
38     end
39 endmodule
  
```

**Codul Verilog** (reluat din imaginea modulului atrib\_bloc-nebloc.v de mai sus) **pentru exemplul de simulare a instrucțiunilor de atribuire *blocking* și *non-blocking*:**

```
module atrib_bloc_nebloc ();
  reg a, b, c, d;
  reg e, f, g, h;

  initial // atribuiri de valori initiale
  begin
    a = 1'b0;
    b = 1'b0;
    c = 1'b0;
    d = 1'b0;

    e <= 1'b0;
    f <= 1'b0;
    g <= 1'b0;
    h <= 1'b0;

    // proceduri de atribuiri blocante
    $display(" atribuiri blocante");
    $display("Time\t a\t b\t c\t d");
    $monitor("%g\t %b\t %b\t %b\t %b", $time, a, b, c, d);
    a = #10 1'b1;
    b = #2 1'b1;
    c = #4 1'b1;
    d = #6 1'b1;
    $display();

    // proceduri de atribuiri neblocante
    $display(" atribuiri neblocante");
    $display("Time\t e\t f\t g\t h");
    $monitor("%g\t %b\t %b\t %b\t %b", $time, e, f, g, h);
    e <= #10 1'b1;
    f <= #2 1'b1;
    g <= #4 1'b1;
    h <= #6 1'b1;
  end
endmodule
```

Mai jos, este prezentat același modul dar cu comentarii adăugate liniilor de program:

```
module atrib_bloc_nebloc ();
  reg a, b, c, d;
  reg e, f, g, h;

  initial // atribuiri de valori initiale
  begin
    a = 1'b0;
    b = 1'b0;
    c = 1'b0;
    d = 1'b0;

    e <= 1'b0;
    f <= 1'b0;
    g <= 1'b0;
    h <= 1'b0;

    // proceduri de atribuiri blocante
    $display(" atribuiri blocante"); //comanda afisarea sirului dintre ghilimele
    $display("Time\t a\t b\t c\t d"); //comanda afisarea etichetelor, tabulate
    $monitor("%2d\t %b\t %b\t %b\t %b", $time, a, b, c, d); //comanda afisarea valorilor

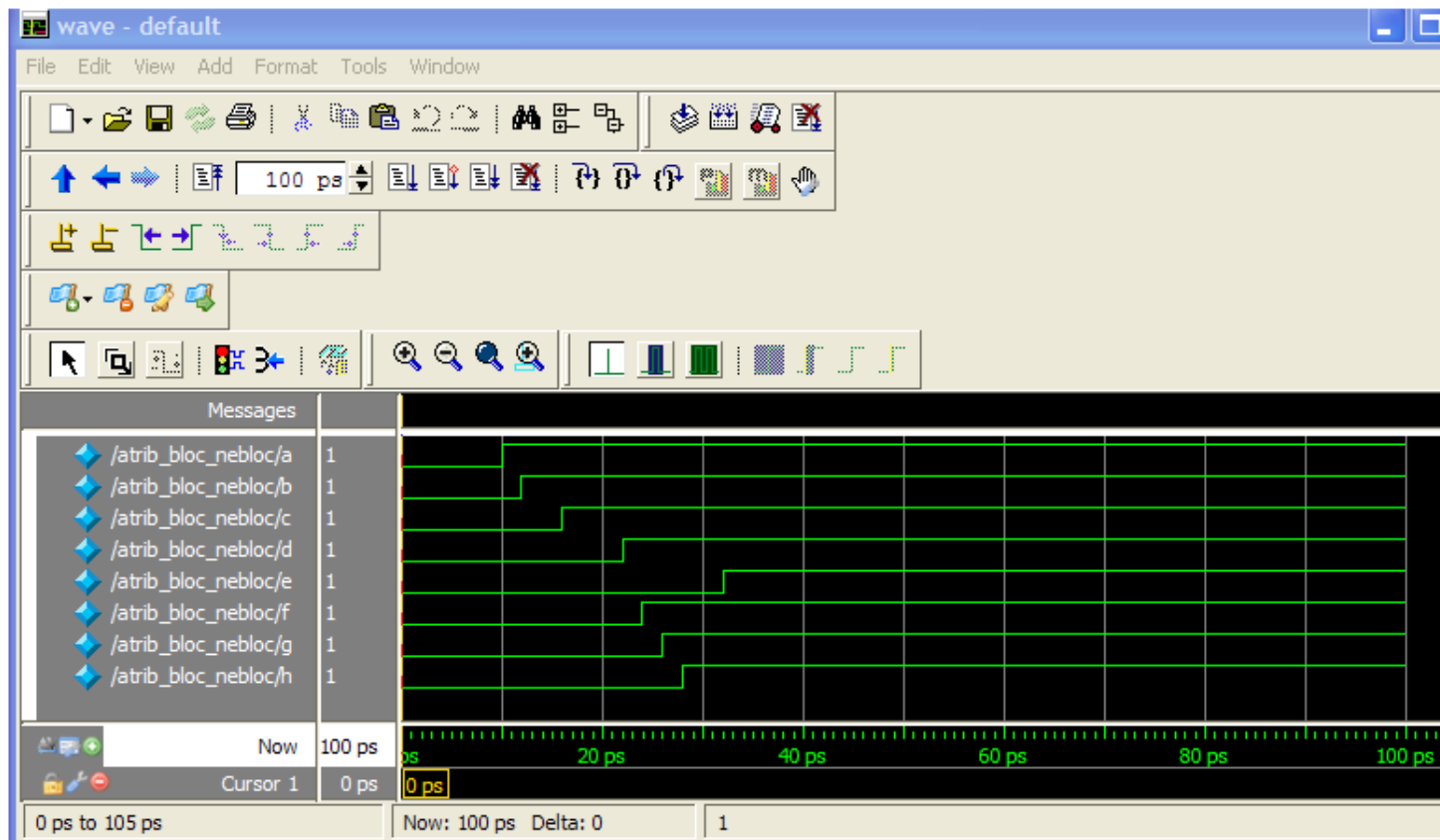
    a = #10 1'b1; // prima atribuire blocanta, la 10 ut de la start simulare
    b = #2 1'b1; // a 2-a atribuire blocanta, la 10+2=12 ut de la start simulare
    c = #4 1'b1; // a 3-a atribuire blocanta, la 12+4=16 ut de la start simulare
    d = #6 1'b1; // a 4-a atribuire blocanta, la 16+6=22 ut de la start simulare
    $display();

    // proceduri de atribuiri nebloante
    $display(" atribuiri nebloante"); //comanda afisarea sirului dintre ghilimele
    $display("Time\t e\t f\t g\t h"); //comanda afisarea etichetelor, tabulate
    $monitor("%2d\t %b\t %b\t %b\t %b", $time, e, f, g, h); //comanda afisarea valorilor

    e <= #10 1'b1; // a 4-a atribuire nebloanta, la 10 ut de la start simulare
    f <= #2 1'b1; // prima atribuire nebloanta, la 2 ut de la start simulare
    g <= #4 1'b1; // a 2-a atribuire nebloanta, la 4 ut de la start simulare
    h <= #6 1'b1; // a 3-a atribuire nebloanta, la 6 ut de la start simulare

  end
endmodule
```

Panourile *wave*, *list* și *Transcript* rezultate de pe urma simulării  
modulului *atrib\_bloc\_nebloc* de mai sus:



```

VSIM 29> run
# atribuiți blocante
# Time  a      b      c      d
# 0      0      0      0      0
# 10     1      0      0      0
# 12     1      1      0      0
# 16     1      1      1      0
#
# atribuiți nebloante
# Time  e      f      g      h
# 22     0      0      0      0
# 24     0      1      0      0
# 26     0      1      1      0
# 28     0      1      1      1
# 32     1      1      1      1
    
```

ps	delta	a	b	c	d	e	f	g	h
0	+0	0	0	0	0	x	x	x	x
0	+1	0	0	0	0	0	0	0	0
10	+0	1	0	0	0	0	0	0	0
12	+0	1	1	0	0	0	0	0	0
16	+0	1	1	1	0	0	0	0	0
22	+0	1	1	1	1	0	0	0	0
24	+0	1	1	1	1	0	1	0	0
26	+0	1	1	1	1	0	1	1	0
28	+0	1	1	1	1	0	1	1	1
32	+0	1	1	1	1	1	1	1	1

Întocmit,  
Coordonator lucr. lab. CLP,  
ing. Hurubeanu Ștefan Valeriu