

2. LIMBAJUL VERILOG HDL

În acest capitol se va prezenta limbajul de descriere arhitecturală Verilog HDL, utilizat în scopul proiectării, implementării, simulării și sintezei logice a unui sistem numeric.

Astfel, vor fi reliefate următoarele aspecte:

- Istoric, caracteristici generale și funcționale, structura și fluxul de proiectare;
- Sintaxa și semantica;
- Operatori, construcții/instrucțiuni, task-uri și funcții;
- Sincronizare și modelare, programe de test.

2.1. INTRODUCERE

2.1.1. ISTORIC

Verilog a fost dezvoltat în momentul când proiectanții căutau unelte ("tools"-uri) "software" pentru a combina diferitele niveluri de simulare. La începutul anilor '1980, existau simulatoare la nivel de comutare, simulatoare la nivel de porți și simulatoare funcționale. Mai mult, majoritatea limbajelor tradiționale de programare erau-sunt în esență secvențiale și astfel semantica lor era o provocare pentru modelarea concurenței din cadrul circuitelor digitale.

Limbajul **Verilog** a fost inventat de Phil Morby de la *Automated Integrated Design Szstems* (redenumită apoi *Gateway Design Automation - GDA*) în anul 1984 ca un limbaj de modelare arhitecturală ("hardware") iar în anul următor a fost scris primul simulator, extins substanțial până în 1987 și redenumit **Verilog-XL**. **Verilog** a împrumutat mult de la limbajele existente, ca de exemplu: aspectele referitoare la concurență de la Modula și Simula, sintaxa de la C și metodele de combinare a nivelurilor de abstractizare de la HiLo (Brunnel University, UK). Limbajul nu era standardizat și a suferit multe modificări până în 1990.

În anul 1989, *GDA* (precum și drepturile asupra **Verilog** și **Verilog-XL**) a fost cumpărată de către *Cadence* care a pus **Verilog** în domeniul public în 1990. Acest lucru a permis și altor companii (precum *Synopsys*) să dezvolte "tools"-uri alternative la Cadence, ceea ce a permis utilizatorilor să adopte limbajul pe scară largă.

Însă, în anul 1981, Departamentul American al Apărării a sponsorizat un “workshop” pe tema limbajelor de descriere “hardware” ca parte a programului VHSIC (“Very High Speed Integrated Circuits”) din care s-au născut specificațiile pentru **VHDL** (Vhsic HDL) în anul 1983. Dezvoltat cu restricții până în anul 1985, **VHDL** a devenit standard IEEE 1076 în anul 1987. Acest lucru a făcut ca, în anul 1990, **Verilog** să devină un limbaj închis. De aceea, Cadence a organizat Open Verilog International (OVI) și a furnizat în anul 1991 documentația pentru **Verilog HDL**. În anul 1992, OVI (cunoscută acum ca *Accellera*) a dus o muncă laborioasă de îmbunătățire a manualului de referință a limbajului (LRM – „Language Reference Manual”) și, în anul 1994, grupul de lucru IEEE 1364 a transformat OVI LRM în standard IEEE.

Astfel, în 1995, **Verilog-HDL** a devenit standard comercial IEEE-1364, fiind referit ca **Verilog-95**. Standardul combina atât sintaxa limbajului **Verilog**, cât și **PLI** (“Programming Language Interface”) într-un singur volum.

În următorii ani au fost adăugate noi caracteristici iar noua versiune a limbajului a devenit standard IEEE 1364-2001 sau **Verilog-2001**. Prin îmbunătățiri ulterioare, printr-un proiect separat System Verilog, limbajul a devenit standard IEEE 1364-2005 sau **Verilog-2005**. Prin includerea suportului de modelare analogice și mixte, limbajul a fost referit ca **Verilog-AMS**. În anul 2005, de către *Co-Design Automation Inc*, s-a dezvoltat un limbaj de verificare de nivel înalt numit **Superlog**; acest limbaj a fost donat către *Accellera*, care l-a transformat în **System Verilog**, devenind standard IEEE P1800-2005 complet aliniat cu **Verilog-2005**.

2.1.2. CARACTERISTICI ȘI STRUCTURA UNUI PROGRAM VERILOG HDL

Verilog este un limbaj de descriere “hardware” (HDL) utilizat pentru a modela sisteme numerice. Limbajul suportă proiectare, verificare și implementare a circuitelor analogice, digitale și mixte pe diferite niveluri de abstractizare.

Limbajul are o sintaxă similară cu cea a limbajului C, ceea ce îl face familiar în utilizare. Astfel, ca și limbajul C, **Verilog** are un pre-procesor, construcții de control ca “if”, “while”, etc, rutine de afișare și operatori similare lui C. El diferă însă fundamental de C în anumite aspecte, ca de exemplu: utilizează begin/end pentru delimitarea blocurilor de cod, utilizează constante definite pe dimensiuni de biți, nu are structuri, pointeri și subrutine recursive (totuși, **System Verilog** include acum aceste capacități) și lucrează cu conceptul de timp, important pentru sincronizare.

Limbajul diferă de un limbaj convențional în sensul că execuția instrucțiunilor nu este strict liniară. Un proiect **Verilog** constă într-o ierarhie de module. Modulele sunt definite ca un set de porturi de intrare, ieșire și bidirecționale. Intern, modulele conțin fire de legătură și registre. Relația dintre porturi, fire și registre este definită prin construcții concurente și secvențiale care stabilesc compartimentul modului. Construcțiile secvențiale sunt plasate în interiorul blocurilor și sunt executate în ordine secvențială în cadrul blocului. Toate construcțiile concurente și blocurile din proiect sunt executate în paralel. Un modul poate conține și una sau mai multe instanțe ale altor module definite în ierarhie.

Un subset de construcții din limbaj sunt sintetizabile. Dacă modulele din proiect conțin numai construcții sintetizabile, programul **Verilog** poate fi utilizat pentru a sintetiza proiectul într-o listă de legături care descrie componentele de bază și conexiunile ce vor fi implementate "hardware". Lista de legături poate fi apoi transformată într-o formă care descrie celule standard ale unui circuit integrat (cum este **ASIC** – "Application Specific Integrated Circuit") sau un flux de biți ("bitstream") pentru un dispozitiv logic programabil (cum este **FPGA** – "Field Programmable Gate Arrays").

Actualmente există o concurență puternică între limbajele **VHDL** și **Verilog**, ceea ce impune prezentarea unei scurte comparații între cele două limbaje.

O prima diferență între limbaje este sintaxa – după cum **Verilog** este bazat pe **C** iar **VHDL** este bazat pe **ADA**:

- **Verilog** este ușor de învățat pentru ca **C** este mai simplu. El produce astfel mai mult cod compact, atât pentru citire, cât și pentru scriere. Mai mult, proiectanții (care deja știu **C** comparativ cu cei care știu **ADA**) îl învață și fac "training" mai ușor.
- **VHDL** este foarte puternic tipizat și permite programatorilor să-și definească propriile lor tipuri deși, în practică se utilizează tipurile de bază și cele definite de IEEE. Beneficiul constă în faptul că verificarea tipului se realizează de către compilator, ceea ce reduce erorile; dezavantajul este că schimbarea tipului trebuie făcută explicit.

Verilog are două avantaje importante față de **VHDL**:

- Permite modelarea la nivel de comutare.
- Se asigură faptul că toate semnalele sunt inițializate ca nedefinite ceea ce asigură că toți proiectanții vor furniza logica necesară de inițializare a proiectelor lor (tipurile de bază din **VHDL** inițializează la 0).

VHDL are două avantaje importante față de **Verilog**:

- Se permit instanțieri condiționale de module (*if/for ... generate*)
- Se furnizează un mecanism simplu (construcția *configure*) care permite proiectantului să comute ușor între descrieri diferite de module particulare.

Alegerea între cele două limbaje nu se poate face însă în mod izolat. Trebuie incluși și alți factori în mediul de proiectare, cum ar fi viteza de simulare, ușurința cu care se poate testa și depana codul, etc.

Verilog include PLI care permite accesul dinamic la structurile de date, ceea ce oferă programatorilor un grad de control mai mare asupra simulării iar proiectanților posibilitatea dezvoltării mediului de proiectare prin includerea de interfețe grafice sau rutine de tip **C** pentru a calcula întârzierile în analiza sincronizării. Pragmatic, un proiectant ar fi bine să le știe pe amândouă.

Stiluri de proiectare

Verilog, la fel ca orice alt limbaj de descriere "hardware", permite realizarea unui proiect în ambele metodologii: "bottom-up" (de jos în sus) sau "top-down" (de sus în jos):

- Proiectarea "bottom-up" este metoda tradițională. Fiecare proiect este realizat la nivel de porți utilizând porți standard, ceea ce este greu de realizat pentru noile structuri numerice care au milioane de tranzistoare. Proiectele astfel realizate sunt grupate la macronivel.

În metodologia "bottom-up", mai întâi se identifică blocurile constructive, care sunt disponibile. În continuare se construiesc celule mai mari utilizând aceste blocuri. Procesul continuă până se ajunge la sinteza blocului de nivel superior.

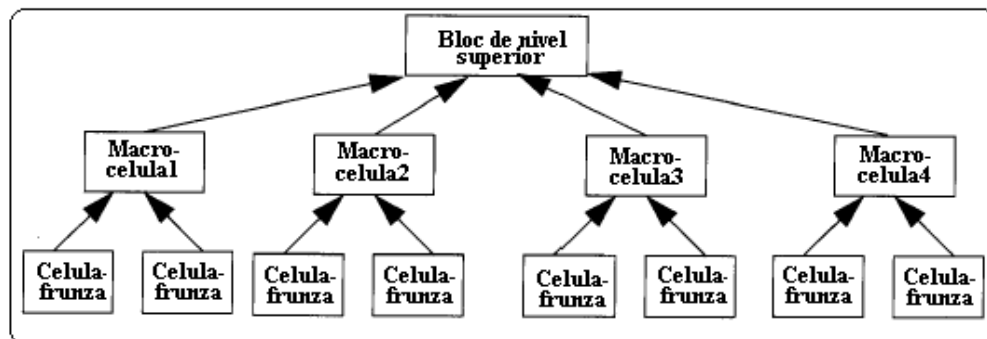


Figura 2.1. Ierarhia de blocuri "bottom-up"

- Proiectarea "top-down" este metoda actuală prin care se pleacă de la specificațiile de sistem și se detaliază în jos până la componentele de sistem. Această metodă permite testarea mai devreme, schimbarea ușoară a tehnologiilor, proiectarea structurată a sistemelor numerice și

multe ale avantajele. Este însă sificil de urmărit un proiect pur “top-down”, de aceea multe proiecte complexe îmbină ambele metode.

În cazul „top-down” se definește un bloc de nivel superior, identificând apoi sub-blocurile necesare pentru construcția acestuia. În continuare sub-blocurile se divid până se ajunge la “celule-frunze”, care nu mai pot fi descompuse în elemente mai simple.

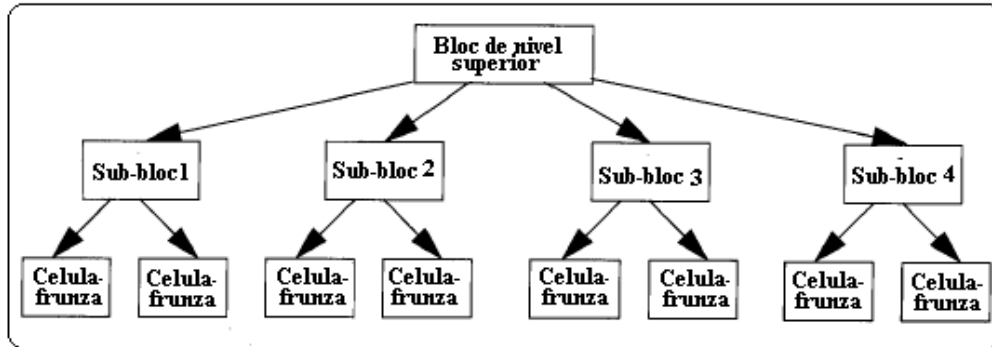


Figura 2.2. Ierarhia de blocuri “top-down”

Structura unui program Verilog

Limbajul Verilog descrie un sistem numeric ca un set de module. Fiecare dintre aceste module are o interfață cu alte module, pentru a specifica maniera în care sunt interconectate. De regula, un modul se plasează într-un fișier, fără ca aceasta să fie o cerință obligatorie. Modulele operează concurrent.

În general, există un modul pe nivelul cel mai înalt, care specifică un sistem închis ce conține, atât datele de test, cât și modelele/modulele hardware. Modulul de pe nivelul cel mai înalt invocă instanțe ale celorlalte module; acest modul poate fi împărțit într-un modul de test și unul sau mai multe module principale, care conțin celelalte submodule.

Modulele reprezintă părți “hardware”, care pot fi de la simple porți până la sisteme complete, ca de exemplu un microprocesor. Ele pot fi specificate, fie comportamental, fie structural (sau o combinație a celor două). O **specificare comportamentală** definește comportarea unui sistem numeric (modul) folosind construcțiile limbajelor de programare tradiționale, de exemplu: **if** și **instrucțiuni de atribuire**. O **specificare structurală** exprimă comportarea unui sistem numeric (modul) ca o conectare ierarhică de submodule.

La baza ierarhiei componentele trebuie să fie primitive sau să fie specificate comportamental.

Primitivele Verilog includ atât porți, cât și tranzistoare de trecere (comutatoare).

Structura unui program este urmatoarea:

- Modul de test;
- Modul (sau module) "top-level";
- Submodul 1;
- ...
- Submodul n;

Fie ca prim exemplu de modul Verilog programul care afișează Hello World (dat ca exemplu în toate limbajele de programare):

```
//-----
// Acesta este primul program Verilog
// Functia : Acest modul va afisa 'Hello World'
//-----
module hello_world ;

initial begin // inceput de bloc de initializare
    $display ("Hello World"); // afisare
    #10 $finish; // terminare afisare
end // sfarsit de bloc

endmodule // sfarsit de modul
```

2.1.3. REPREZENTAREA CIRCUITELOR ȘI SITEMELOR NUMERICE – NIVELURI DE ABSTRACTIZARE VERILOG

Printr-o abordare ierarhică, un sistem "hardware" complex poate fi descris sub aspect comportamental, structural, fizic și abstract pe mai multe niveluri:

| Modalități de descriere: | Comportamentală | Structurală | Fizică | Abstractă |
|--------------------------|--------------------------------|----------------------------|--------------------------|--|
| Niveluri ierarhice: | Aplicații, sistemul de operare | Calculator, procesor | Modul, plachetă, circuit | Arhitectură, algoritm, modul/bloc funcțional |
| | Programe | Sumatoare, porți, registre | Celule | Logică, comutație |
| | Subrutine, instrucțiuni | Tranzistoare | Tranzistoare | Circuit |

Abordarea pe niveluri de abstractizare ajută la alegerea unui stil de proiectare, în funcție de complexitatea structurilor numerice. În continuare se definesc aceste niveluri:

1. Arhitectura, nivelul cel mai de sus, se referă la aspectele funcționale ale sistemului. La acest nivel există următoarele trei subniveluri:
 - arhitectura sistemului: subnivelul cel mai înalt, care constituie interfața cu utilizatorul;
 - organizarea sistemului: structurarea sistemului în unități funcționale, interconectarea lor, fluxul de informații dintre ele;
 - microarhitectura: structura unităților funcționale, care realizează interfața dintre “hardware” și “firmware”.
2. Algoritmul mapează nivelul arhitectural la nivelul logic.
3. Modulul/blocul funcțional este un nivel tipic de transfer între registre (RTL – “Register Transfer Level”), care caracterizează în termeni ca registre, întârzieri, numărătoare, ceas, memorie, circuite combinaționale și secvențiale.
4. Logica reprezintă nivelul care descrie sistemul, utilizând porți logice.
5. Comutația este nivelul care analizează comutarea semnalelor de pe un nivel logic pe altul.
6. Circuitul constituie nivelul abstract cel mai scăzut, care reprezintă sistemul la nivel de circuite electronice.

Inițial descrierea structurilor hardware a fost realizată la nivelul transferurilor între registre (sunt cunoscute limbaje ca APL (“A Programming Language”) sau AHPL (“A Hardware Programming Language”)); ele pot servi într-o mai mică măsură și la nivelul proiectării logice.

Începând cu anii ‘1980 au apărut limbaje pentru descrierea la nivel arhitectural, cum ar fi S*M sau ADL.

Pentru a evita inconvenientul major de descriere a structurilor “hardware”, utilizând mai multe limbaje adecvate diferitelor niveluri prezentate mai sus, s-a căutat dezvoltarea limbajelor pentru a obține unele capabile să înțeleagă cât mai multe niveluri de abstractizare, mergând până la nivelul logic.

Primele dintre ele au fost MIMOLA și M, dar cele care reprezintă la ora actuală vârful în domeniul proiectării automate a structurilor “hardware” sunt VHDL (“Very High Speed Integrated Circuit Hardware Description Language”) și Verilog HDL.

Niveluri de abstractizare Verilog

Verilog suportă proiectare pe diferite niveluri de abstractizare. Cele mai importante sunt nivelul comportamental, nivelul RTL, nivelul structural sau poartă și nivelul fizic.

- **Nivelul comportamental**

Nivelul comportamental descrie un sistem prin algoritmi concurenți. Fiecare algoritm este el însuși secvențial, ceea ce înseamnă că el constă dintr-un set de instrucțiuni care se execută una după cealaltă. Elementele principale sunt funcțiile, "task"-urile și blocurile "initial" și "always".

Exemplu:

```
primitive dEdgeFF_UDP (q, clk, d);
  output q;
  reg    q;
  input  clk, d;
  table
//      clk      d      stare    q
      (01)      0      : ? :    0;
      (01)      1      : ? :    1;
      (0x)      1      : 1 :    1;
      (0x)      0      : 0 :    0;
      (?0)      ?      : ? :    -;
      ?         (??)   : ? :    -;
  endtable
endprimitive
```

- **Nivelul RTL**

Proiectarea la nivel RTL specifică toate caracteristicile unui circuit prin operații și transferuri de date între registre. Se utilizează un semnal explicit de ceas. Proiectele RTL conțin delimitări exacte de întârzieri: operațiile sunt planificate să apară la anumite momente de timp. O definiție modernă a codului RTL este: Orice cod care este sintetizabil este numit cod RTL.

Exemplu:

```
module dEdgeFF_RTL (q, clk, d);
  output q;
  reg    q;
  input  clk, d;

  initial
    q = 0;
  always @(negedge clk)
    #10 q = d;
endmodule
```


- **Nivelul poartă (structural)**

La nivel poartă logică, caracteristicile unui sistem sunt descrise prin semnale logice și întârzierile lor. Toate semnalele pot avea doar valori logice de tipul "0", "1", "X" și "Z". Operațiile uzuale sunt predefinite ca primitive logice (porți AND, OR, NOT, etc). Utilizarea modelării la nivel de poartă poate să nu fie cea mai bună idee pentru orice nivel al proiectării logice. Codul de nivel poartă este generat de "tool"-uri precum cel de sinteză iar lista de legături generată este utilizată pentru simularea la nivel de porți.

Exemplu:

```
module dEdgeFF_gates(d,clk,q,q_bar);
input d,clk;
output q, q_bar;

wire n1,n2,n3,q_bar_n;
wire cn,dn,n4,n5,n6;

// primul "latch"
not (n1,d);

nand (n2,d,clk);
nand (n3,n1,clk);

nand (dn,q_bar_n,n2);
nand (q_bar_n,dn,n3);

// al doilea „latch”
not (cn,clk);
not (n4,dn);

nand (n5,dn,clk);
nand (n6,n4,clk);

nand (q,q_bar,n5);
nand (q_bar,q,n6);

endmodule
```

- **Nivelul fizic (al tranzistoarelor)**

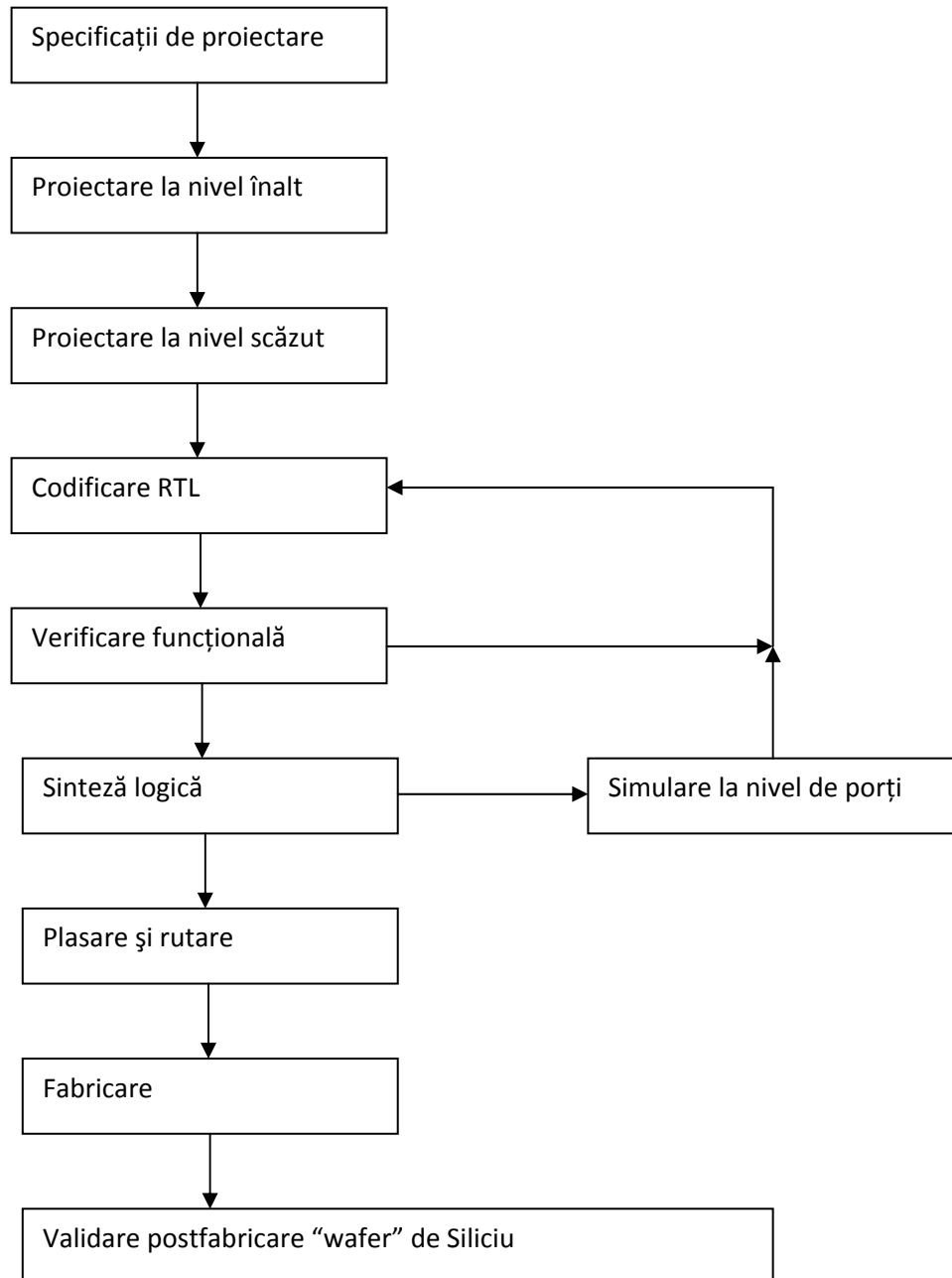
Descrierea fizică furnizează informații privind modul de construcție al unui circuit particular descris din punct de vedere comportamental și structural; descrierea fizică este asociată cu descrierea la nivel de măști tehnologice. Această descriere cuprinde mai multe niveluri de abstractizare: nivelul modul (descrie limitele geometrice externe pentru circuit), un set de submodule (librării de primitive fizice), nivelul porturilor (corespund unei conexiuni de intrare/ieșire din descrierea structurală a

componentei pentru care se specifică poziția, stratul, numele și lățimea) iar la nivelul cel mai de jos se află apelările de tranzistoare, fire și la conexiuni.

În continuare se va considera doar proiectarea la nivel RTL și comportamental.

2.1.4. FLUXUL DE PROIECTARE

Fluxul tipic de proiectare a unui sistem numeric utilizând limbajul **Verilog** poate fi reliefat astfel:



Specificatii de proiectare

Este etajul în care se definesc parametrii de proiectare sistem. Astfel, se specifică scopul proiectării, modurile de funcționare, tipurile de date și funcții, etc. În acest scop se pot folosi editoare de texte.

Proiectarea la nivel înalt

Acesta este etajul în care se precizează blocurile funcționale ale sistemului de proiectat, intrările și ieșirile în/din sistem, forme de undă precum și modul de comunicare dintre blocurile funcționale. În acest scop se pot folosi editoare de texte și de forme de undă.

Proiectarea la nivel scăzut

Proiectarea aceasta se mai numește și microproiectare și este faza în care proiectantul descrie implementarea fiecărui bloc. Ea conține detalii despre mașina de tranziții de stări și despre blocurile funcționale. Este bine să se deseneze formele de undă pentru diferitele interfațări. La această fază se pierde destul de mult timp. Se pot utiliza editoare de texte și de forme de undă.

Codificarea RTL

În această fază, microproiectarea este convertită în cod **Verilog** utilizând constructorii sintetizabili ai limbajului. Se utilizează editoare HDL separate sau cele din cadrul “tool”-urilor folosite la proiectare.

Verificarea și simularea

Această etapă este procesul de verificare funcțională la orice nivel de abstractizare. Se utilizează simulatoare adecvate. Pentru a testa dacă codul RTL îndeplinește cerințele funcționale din specificațiile de proiectare, trebuie să se scrie programe de test (“testbench”) cu care se va verifica funcțional codul RTL proiectat pe baza studiului formelor de undă obținute în urma simulării. Se pot utiliza simulatoare/compileroare separate sau incluse în diferite “tool”-uri de proiectare, precum Modelsim, ISE Xilinx, Veriwell, Icarus, etc. Pentru a verifica simularea întârzierilor pentru sincronizare (atât la nivel de porți, cât și la nivel de fire), după faza de sinteză, se mai realizează o simulare la nivel de porți sau SDF (“Standard Delay Format”).

Sinteza

Este procesul prin care “tool”-urile de sinteză ca XST, Ahdl sau Synplify iau codul RTL și, ținând cont de tehnologie și constrângeri ca intrări, îl mapează în primitive tehnologice la nivel de porți. “Tool”-ul de

sinteză, după mapare, realizează și o analiză minimală a întârzierilor pentru a verifica cerințele de sincronizare (însă doar la nivel de porți, la nivelul firelor se face în faza de plasare și rutare).

Plasarea și rutarea

Lista de legături la nivel de porți generată de “tool”-ul de sinteză este luată și importată în “tool”-ul de plasare și rutare în format listă de legături **Verilog**. Mai întâi toate porțile și bistabilii sunt plasați iar semnalele de ceas și inițializare sunt rutate, după care fiecare bloc este rutat obținându-se ca ieșire un fișier GDS utilizat la fabricarea circuitelor ASIC.

Validare postfabricare

O dată ce circuitul este gata de fabricație, este necesar ca el să fie pus în condiții de mediu real de funcționare și testat înainte de a fi lansat pe piață. Aceasta deoarece viteza de simulare a RTL este scăzută și există posibilitatea găsirii unor “bug”-uri la testarea “wafer”-elor de Siliciu.

2.2. SINTAXA ȘI SEMANTICA LIMBAJULUI VERILOG HDL

2.2.1. CONVENȚII LEXICALE

Sunt similare cu cele ale limbajului **C. Verilog** este un limbaj “case-sensitive” în care toate cuvintele cheie sunt cu litere mici.

Spații libere (albe)

Spațiile albe pot conține caractere precum spații (/b – “blank”), caractere “tab” (/t), caractere linie nouă (/n), caractere sfârșit de linie (“Carriage Return” - <CR>), sfârșit de paragraf și sfârșit de fișier (EOF – “End Of File”).

Aceste caractere sunt ignorate cu excepția când servesc la separarea altor convenții sau în cadrul șirurilor.

Comentarii

Există două moduri de a introduce comentarii:

- Comentarii pe o singură linie, care încep cu // și sfârșesc cu <CR>
- Comentarii pe linii multiple, care încep cu /* și sfârșesc cu */

Identificatori

Identificatorii sunt nume date pentru un obiect, precum un registru, o funcție sau un modul, astfel încât să poată fi referit din alte părți ale descrierii RTL.

Identificatorii trebuie să înceapă cu un caracter alfabetic sau caracterul “underscore” (_). Ei pot conține caractere alfabetice, caractere numerice, _ și \$. De asemenea, ei pot avea o lungime de până la 1024 caractere.

Verilog permite utilizarea oricărui alt caracter prin omiterea lui plasând un caracter \ (“backslash”) la începutul identificatorului și terminând numele lui cu un spațiu liber.

De exemplu:

```
module \1dEdge_FF (... , \q~, \reset*, ...);  
...
```

Valori logice

În **Verilog** există trei tipuri de valori logice:

- 0 – zero, nivel scăzut sau fals
- 1 – unu, nivel înalt sau adevărat
- Z sau z – înaltă impedanță (valoare trei-stări sau flotantă)
- X sau x - nedefinit, necunoscut sau neinițializat

Puteri logice

În **Verilog** există următoarele tipuri de puteri logice (valori logice puternic cuplate – “logic strengths”):

| Nivel de putere | Nume valoare puternic cuplată | Mnemonică specifică | | Afișare mnemonică | |
|-----------------|-------------------------------|---------------------|---------|-------------------|------|
| 7 | “Drive” de alimentare | supply0 | supply1 | Su0 | Su1 |
| 6 | “Drive” puternic cuplat | strong0 | strong1 | St0 | St1 |
| 5 | “Drive” ținut („pull”) | pull0 | pull1 | Pu0 | Pu1 |
| 4 | Capacitate mare | Large | | La0 | La1 |
| 3 | “Drive” slab cuplat | weak0 | weak1 | We0 | We1 |
| 2 | Capacitate medie | Medium | | Me0 | Me1 |
| 1 | Capacitate mică | Small | | Sm0 | Sm1 |
| 0 | Impedanță înaltă | highz0 | highz1 | HiZ0 | HiZ1 |

2.2.2. REPREZENTAREA NUMERELOR

Se pot specifica constante numerice în format zecimal, hexazecimal, octal sau binar. **Verilog** suportă atât numere cu semn, cât și numere fără semn; orice număr fără semnul “-” este considerat pozitiv în timp ce numerele precedate de “-” sunt considerate negative cu semn. Numerele negative se reprezintă în cod complementar.

Numere întregi

Sintaxa este:

<mărime>'<bază> <valoare>

Caracteristici:

- Numerele sunt cu semn sau fără semn. Între mărime, bază și valoare sunt permise spații
- <bază> (opțional) este baza de numerație (binară, octală, zecimală sau hexazecimală). Ea este predefinită ca zecimală și nu este “case-sensitive”
- <mărime> (opțional) este numărul de biți din întreg. Întregii fără semn sunt predefiniți la 32 biți
- <valoare> nu este “case-sensitive” iar ? este un alt mod de a reprezenta valoarea Z

| Baza | Simbol | Valori legale |
|-------------|---------|---------------------------------|
| Binar | b sau B | 0, 1, x, X, z, Z, ?, _ |
| Octal | o sau O | 0-7, x, X, z, Z, ?, _ |
| Zecimal | d sau D | 0-9, _ |
| Hexazecimal | h sau H | 0-9, a-f, A-F, x, X, z, Z, ?, _ |

- Caracterul _ (“underscore”) poate fi pus oriunde în interiorul numărului și este ignorat
- Valorile sunt expandate de la dreapta la stânga (lsb (“least significant bit”) spre msb (“most significant bit”))
- Când mărimea este mai mică decât valoarea, atunci biții cei mai semnificativi sunt trunchiați. Când mărimea este mai mare decât valoarea, atunci biții cei mai semnificativi sunt setați la 0 dacă valoarea este 0 sau 1, la Z sau X dacă valoarea este Z sau X
- Valorile ‘0’ și ‘1’ pot fi folosite ca numere pentru că nu pot fi identificatori
- Numerele pot fi declarate ca parametri

Numere reale

Sintaxa este:

<valoare>.<valoare> (notația zecimală) sau

<valoare mantisă>E<valoare exponent> unde E nu este "case-sensitive" (notația științifică)

Caracteristici:

- **Verilog** suportă constante și variabile reale și convertește numerele reale la întregi prin rotunjire
- Numerele reale nu pot conține Z sau X și pot avea valori doar de la 0 la 9 și _ doar în valoare
- Numerele reale sunt rotunjite la cel mai apropiat întreg când sunt atribuite la un întreg

2.2.3. MODULE; INSTANȚIERI, PARAMETRIZĂRI ȘI IDENTIFICATORI IERARHICI

Definirea modulelor în Verilog HDL

Modulele sunt blocurile constructive ale proiectelor **Verilog**. Se poate crea un proiect ierarhic prin instanțierea modulelor în alte module. Un modul este instanțiat când se utilizează acest modul în altul, pe un nivel ierarhic superior.

Sintaxa unui modul este următoarea:

| Sintaxa |
|--|
| <p style="text-align: center;">Conexiune internă implicit (prin ordinea porturilor)</p> <pre>module nume_modul (nume_port, nume_port, ...); componente_modul endmodule</pre> |
| <p style="text-align: center;">Conexiune internă explicit (prin nume)</p> <pre>module nume_modul (.nume_port (nume_semnal), .nume_port (nume_semnal), ...); componente_modul endmodule</pre> |

- Conexiunea internă implicită conectează portul la un fir intern sau un registru cu același nume. Ordinea trebuie să se potrivească corect. Normal, nu este o idee bună să se conecteze porturile implicit pentru că ar putea cauza probleme în depanare (de exemplu localizarea unui port care a cauzat o eroare la compilare), când orice port este adăugat sau șters.
- Conexiunea internă explicită conectează portul la un semnal intern cu nume diferit sau la semnale interne selectate pe biți sau pe părți sau concatenate. Numele trebuie să se potrivească cu modulul frunză iar ordinea nu este importantă.
- Componentele unui modul sunt:
 - declarații_porturi_modul
 - declarații_tipuri de date
 - instanțieri_module
 - instanțieri_primitive
 - blocuri_procedurale
 - atribuiri_continue
 - definiții_task-uri
 - definiții_funcții
 - blocuri_specificate
- Funcționalitatea unui modul poate fi:
 - Comportamentală (RTL): modelat cu constructori de blocuri procedurale sau atribuiri continue
 - Structurală: modelat ca o listă de legături de instanțe de module și/sau primitive
 - Combinată: comportamental și structural
- Definițiile de module nu pot fi îmbricate însă un modul poate instanția alte module.
- Definițiile de module pot fi exprimate utilizând constante parametru. Fiecare instanță de modul poate redefini parametrii pentru a fi unici pentru acea instanță.
- Componentele unui modul pot apărea în orice ordine dar declarațiile de porturi și cele de tipuri de date trebuie să apară înainte ca porturile sau semnalele să fie referite.

Porturi

Porturile permit comunicarea între un modul și componentele sale. Toate modulele mai puțin modulul de testare au porturi. Porturile pot fi asociate prin ordine sau prin nume.

Sintaxa de declarare a porturilor este următoarea:

| Sintaxa |
|--|
| <code>direcție_port [msb : lsb] nume_port, nume_port, ... ;</code> |

- Direcția porturilor poate fi:
 - **input** pentru scalari sau vectori de intrare
 - **output** pentru scalari sau vectori de ieșire
 - **inout** pentru scalari sau vectori bidirecționali
- Valorile msb și lsb trebuie să fie întregi, parametri întregi sau o expresie care se rezumă la o constantă întreagă.
Se poate folosi atât convenția “little-endian” (lsb este cel mai mic bit) cât și convenția “big-endian” (lsb este cel mai mare bit)
- Dimensiunea maximă a unui port poate fi limitată dar nu mai puțin de 256 biți
- Ca o practică bună, se recomandă declararea unui singur port de identificare pe linia program pentru a putea fi comentat
- Reguli de conectare a porturilor:
 - Intrările: intern trebuie să fie întotdeauna un tip de fir, extern pot fi conectate la o variabilă de tip registru sau fir
 - Ieșirile: intern pot fi de tip fir sau registru, extern trebuie să fie conectate la o variabilă de tip registru
 - Porturile bidirecționale: intern și extern trebuie întotdeauna să fie de tip fir și pot fi conectate numai la o variabilă de tip fir
 - Potrivirea ca dimensiuni: este legal să se conecteze intern și extern porturi de dimensiuni diferite; dar atenție că tool-urile de sinteză pot raporta probleme
 - Porturi neconectate: sunt permise prin utilizarea notației “,”
 - Firele tipuri de date sunt utilizate pentru a conecta structura; un astfel de fir este cerut dacă un semnal poate fi dat într-o conexiune structurală

Instantierea modulelor

Sintaxa pentru instanțiere este următoarea:

| Sintaxa |
|---|
| <p align="center">Conexiuni implicite prin ordinea porturilor</p> <p>nume_modul nume_instanță [domeniu_tablou_instanțe] (semnal, semnal, ...);</p> |
| <p align="center">Conexiuni explicite prin numele porturilor</p> <p>nume_modul nume_instanță [domeniu_tablou_instanțe] (.nume_port(semnal), (.nume_port(semnal), ...);</p> |
| <p align="center">Redefinire explicit prin parametri</p> <p>defparam cale_ierarhică.nume_parametru = valoare;</p> |
| <p align="center">Redefinire implicit prin parametri</p> <p>nume_modul #(valoare sau nume parametru) nume_instanță (semnale);</p> |

- Un modul poate fi instanțiat utilizând ordinea sau numele porturilor:
 - Instanțierea prin ordinea porturilor listează semnalele conectate la ele în aceeași ordine ca în lista de porturi din definirea modului
 - Instanțierea prin numele porturilor listează numele portului și semnalul conectat la el în orice ordine
- *Nume_instanță* (cerut) este utilizat pentru a realiza instanțe multiple ale aceluiași modul în mod unic dintr-un altul.
- *Domeniu_tablou_instanțe* (opțional) instanțiază module multiple, fiecare instanță fiind conectată la biți separați ai unui vector:
 - Domeniul este specificat ca [lhi:rho] (index stânga ("left-hand-index") la index dreapta ("right-hand-index"))
 - Dacă dimensiunea pe biți a unui modul port dintr-un tablou este aceeași cu cea a semnalului conectat la el, semnalul complet este conectat la fiecare instanță a modului port; dacă dimensiunea diferă, atunci fiecare instanță de modul port este conectată la o parte selectată a semnalului, cu cel mai puțin semnificativ index de instanță conectat la cea mai puțin semnificativă parte a unui vector și tot așa progresiv către stânga
 - Trebuie să existe un număr corect de biți în fiecare semnal pentru a se conecta la toate instanțele (mărimea semnalului și mărimea portului trebuie să fie multiple)

- Parametrii dintr-un modul pot fi redefiniți pentru fiecare instanță:
 - Redefinirea explicită utilizează o construcție **defparam** cu numele ierarhic al parametrului
 - Redefinirea implicită utilizează jetonul # ca parte a instanțierii modulului. Parametrii trebuie redefiniți în aceeași ordine în care au fost declarați prin modul
- Numele căii ierarhice este dat de identificatorul modulului rădăcină (“top-level”) urmat de identicatorii instanțelor modulului, separate prin puncte. Acest lucru este folositor atunci când se dorește a se vedea semnalul dintr-un modul frunză sau când se dorește forțarea unei valori înăuntru unui modul intern.

2.2.4. PRIMITIVE PROGRAM

Primitive

Verilog are primitive incluse (“built-in”) cum sunt porțile, porțile de transmisie și comutatoarele. Acestea sunt utilizate rar în proiecte RTL, dar sunt utilizate în faza de postsinteză pentru modelarea celulelor ASIC/FPGA; aceste celule sunt apoi utilizate pentru simularea la nivel de porți, sau cum se mai numește, simularea SDF (“Standard Delay Format”). De asemenea, formatul de ieșire al listei de legături din tool-ul de sinteză, care este importat în tool-ul de plasare și de rutare, este tot lcu primitive **Verilog** la nivel de porți. Sintaxa de utilizare a acestor primitive este următoarea:

| Poartă | Descriere | List de terminale |
|--|---|-------------------------------------|
| and or xor | Poartă ȘI Poartă SAU Poartă SAU-EXCLUSIV | (1 ieșire, 1 sau mai multe intrări) |
| nand nor xnor | Poartă ȘI-NU Poartă SAU-NU Poartă SAU-NU-EXCLUSIV | (1 ieșire, 1 sau mai multe intrări) |

| Poartă de transmisie | Descriere | List de terminale |
|--------------------------------|---|-------------------------------------|
| not buf | Inversor Tampon | (1 sau mai multe ieșiri, 1_intrare) |
| bufif0 bufif1 | Tampon cu trei stări (Low) Tampon cu trei stări (High) | (1-ieșire, 1-intrare, 1-control) |
| notif0 notif1 | Inversor cu trei stări (Low) Inversor cu trei stări (High) | (1-ieșire, 1-intrare, 1-control) |

| Comutatoare | Descriere | List de terminale |
|-----------------------------------|---|---|
| pmos rpmos | PMOS unidirecțional PMOS rezistiv | (1-ieșire, 1-intrare, 1-control) |
| nmos rnmos | NMOS unidirecțional NMOS rezistiv | (1-ieșire, 1-intrare, 1-control) |
| cmos rcmos | CMOS unidirecțional CMOS rezistiv | (1-ieșire, 1-intrare, n-control, p-control) |
| tranif1 rtranif1 | Tranzistor bidirecțional (High) Tranzistor rezistiv (High) | (2-bidirecționale, 1-control) |
| tranif0 rtranif0 | Tranzistor bidirecțional (Low) Tranzistor rezistiv (Low) | (2-bidirecționale, 1-control) |
| tran rtran | Tranzistor de trecere bidirecțional Tranzistor de trecere rezistiv | (2-bidirecționale) |
| pullup pulldown | Rezistor la sursă Rezistor la masă | (1-ieșire) |

Instantierea primitivelor

Sintaxa de instanțiere este următoarea:

| Sintaxă primitivă |
|--|
| tip_poartă (<i>drive_strength</i>) #(întârziere) nume_instanță [domeniu_instanță_tablou] (<i>terminal, terminal, ...</i>); |
| tip_comutator #(întârziere) nume_instanță [domeniu_instanță_tablou] (<i>terminal, terminal, ...</i>); |

iar sintaxa întârzierilor primitivelor este:

| Sintaxă întârziere primitivă |
|---|
| #întârziere or #(întârziere) <i>Întârziere singulară pentru toate tranzițiile de ieșire</i> |
| #(întârziere, întârziere) <i>Întârzieri separate pentru tranziții (rising, falling)</i> |
| #(întârziere, întârziere, întârziere) <i>Întârzieri separate pentru tranziții (rising, falling, turn-off)</i> |
| #(întârziere_minimă:întârziere_tipică:întârziere_maximă) <i>Domeniu de întârzieri de la minimum la maximum pentru toate tranzițiile</i> |

```
#(întârziere_minimă:întârziere_tipică:întârziere_maximă,  
întârziere_minimă:întârziere_tipică:întârziere_maximă)  
Domeniu de întârzieri de la minimum la maximum pentru tranziții (rising, falling)
```

```
#(întârziere_minimă:întârziere_tipică:întârziere_maximă,  
întârziere_minimă:întârziere_tipică:întârziere_maximă,  
întârziere_minimă:întârziere_tipică:întârziere_maximă)  
Domeniu de întârzieri de la minimum la maximum pentru tranziții (rising, falling, turn-off)
```

unde:

- Întârzierea “rise” este asociată tranziției ieșirii porții în 1 din 0, x, z
- Întârzierea “fall” este asociată tranziției ieșirii porții în 0 din 1, x, z
- Întârzierea “turn-off” este asociată tranziției ieșirii porții în z din 0, 1, x
- Întârzierea (opțional) reprezintă timpul de propagare prin primitivă și este predefinită la zero; poate avea valori întregi sau reale
- Nume_instanță (opțional) poate fi utilizat pentru a referi primitive specifice în tool-urile de depanare, scheme, etc
- Domeniu_instanță_tablou (opțional) instanțiază primitive multiple, fiecare instanță fiind conectată la biți separați ai unui vector, unde semnalele vector trebuie să aibă aceeași dimensiune cu cea a tabloului iar semnalele scalare sunt conectate la toate instanțele din tablou; domeniul este specificat ca [lhi:rho] la fel ca la instanțierea modulelor

Primitive Definite de Utilizator

Cum primitivele incluse în **Verilog** sunt însă puține, dacă este nevoie de primitive mai complexe atunci se furnizează UDP (“User Defined Primitives” – primitive definite de utilizator).

UDP pot modela logică combinațională și secvențială și pot include întârzieri. Ele trebuie definite în afara corpului **module ... endmodule**.

Sintaxan UDP este următoarea:

primitive nume_primitivă (porturi/terminale);

declarații_primitivă ...

construcții_initial ...

tabelă_definiție

endprimitive

unde:

- nume_primitivă: identificator
- declarații_UDP: declarații_ieșiri_UDP, declarații_registre pentru ieșiri (numai pentru UDP secvențiale), declarații_intrări_UDP;
- construcții **initial** (opțional): stabilesc valoarea inițială (1'b0, 1'b1, 1'bx, 1, 0) a terminalelor de ieșire prin atribuirea unei valori literale pe un singur bit unui registru terminal de ieșire; valoarea predefinită este "x".
- tabela_definiție a funcționării: descrie funcționalitatea primitive, atât combinațional, cât și secvențial și are structura **table ... intrări_tabelă ... endtable**;

Caracteristicile UDP:

- Reguli de folosire a porturilor în UDP:
 - O UDP poate conține numai o ieșire și până la 9 intrări (UDP secvențiale) sau 10 intrări (UDP combinaționale); portul de ieșire trebuie să fie primul port urmat de una sau mai multe porturi de intrare
 - Toate porturile UDP sunt scalari; porturile vectori nu sunt permise; UDP nu pot avea porturi bidirecționale
 - Terminalul de ieșire a unei UDP secvențiale cere o declarație suplimentară ca tip registru
 - Este ilegal să se declare un registru pentru un terminal de ieșire a unei UDP combinaționale
- Fiecare linie dintr-tabelă este o condiție; când o intrare se schimbă, condiția de intrare este marcată și ieșirea este evaluată pentru a reflecta noua schimbare a intrării
- UDP utilizează simboluri speciale pentru a descrie funcții ca front crescător, nu contează ș.a:

| Simbol | Interpretare | Explicație |
|---------------------|--|---|
| 0 sau 1 sau x sau X | 0 sau 1 sau x sau X | 0, 1 sau necunoscut pe intrare sau ieșire |
| ? | 0 sau 1 sau x | Intrarea poate fi 0 sau 1 sau x |
| b sau B | 0 sau 1 | Intrarea poate fi 0 sau 1 |
| f sau F | (10) | Front descrescător pe o intrare |
| r sau R | (01) | Front crescător pe o intrare |
| p sau P | (01) sau (0x) sau (x1) sau (1z) sau (z1) | Front pozitiv incluzând x și z |
| n sau N | (10) sau (1x) sau (x0) sau (0z) sau (z0) | Front negativ incluzând x și z |
| * | (??) | Toate tranzițiile posibile pe intrări |
| - | Nici o schimbare | Nici o schimbare la ieșire (numai pentru UDP secvențiale) |

- **UDP combinaționale**

Ieșirea este determinată ca o funcție de intrarea curentă. Ori de câte ori o intrare își schimbă valoarea, UDP este evaluată și unul dintre rândurile tabeli de stare este marcat. Ieșirea stării este setată la valoarea indicată prin acel rând. Acest lucru este similar construcțiilor de condiție: fiecare linie din tabelă este o condiție. UDP combinaționale au un câmp pentru intrări și unul pentru ieșire, care sunt separate prin simbolul ":". Fiecare rând al tabeli este terminat prin simbolul ";" . Ordinea intrărilor în descrierea tabeli de stare trebuie să corespundă cu ordinea intrărilor din lista de porturi din antetul de definiție a UDP. Nu are legătură cu ordinea de declarare a intrărilor. Intrările trebuie separate unele de altele printr-un spațiu liber.

Fiecare rând din tabelă definește ieșirea pentru combinație particulară de stări de intrare. Dacă toate intrările sunt specificate ca "x", atunci ieșirea trebuie specificată ca "x". Toate combinațiile care nu sunt specificate explicit rezultă în starea predefinită de ieșire ca "x".

Este ilegal să existe aceeași combinație de intrări, specificată pentru diferite ieșiri.

- **UDP secvențiale senzitive pe nivel**

Comportarea este specificată la fel ca mai sus, cu excepția faptului că ieșirea este declarată ca tip registru și există un câmp suplimentar în fiecare intrare a tabeli, care reprezintă starea curentă a UDP.

Ieșirea indică faptul că există o stare internă și are aceeași valoare cu ea. Câmpul suplimentar este separat prin simbolul ":" între câmpul pentru intrări și cel pentru ieșire.

- **UDP secvențiale senzitive pe front**

Diferă față de cele anterioare prin faptul că schimbările la ieșire sunt eșantionate prin tranziții specifice ale intrărilor. Pentru fiecare intrare în tabelă, numai un singur semnal de poate avea poate avea specificată o tranziție pe front. Toate celelalte semnale de intrare trebuie să aibă intrări în tabelă pentru a acoperi tranzițiile sau UDP va furniza "x" la ieșire când tranziția apare. Intrările din tabelă senzitive pe front sunt mai puțin prioritare decât cele senzitive pe nivel.

2.2.5. TIPURI DE DATE; ȘIRURI

Sintaxa de declarare a tipurilor de date este următoarea:

| Sintaxa |
|---|
| tip_registru [dimensiune] <i>nume_variabilă</i> , <i>nume_variabilă</i> , ... ; |
| tip_registru [dimensiune] <i>nume_memorie</i> [dimensiune_tablou]; |
| tip_legătură [dimensiune] #(întârziere) <i>nume_legătură</i> , <i>nume_legătură</i> , ... ; |
| tip_legătură ("drive_strength") [dimensiune] #(întârziere) <i>nume_legătură</i> = atribuire_continuă; |
| trireg ("capacitance_strength") [dimensiune] #(întârziere, timp_decay) <i>nume_legătură</i> , <i>nume_legătură</i> , ... ; |
| parameter <i>nume_constantă</i> = <i>valoare</i> , <i>nume_constantă</i> = <i>valoare</i> , ... ; |
| specparam <i>nume_constantă</i> = <i>valoare</i> , <i>nume_constantă</i> = <i>valoare</i> , ... ; |
| event <i>nume_eveniment</i> , <i>nume_eveniment</i> , ... ; |

unde:

- Întârziere (opțional) poate fi specificată numai pentru tipul legătură. Sintaxa este la fel ca la primitive.
- Dimensiunea este un domeniu de forma [msb : lsb].
Valorile msb și lsb trebuie să fie întregi, parametri întregi sau o expresie care se rezumă la o constantă întreagă. Se poate folosi atât convenția "little-endian" (lsb este cel mai mic bit) cât și convenția "big-endian" (lsb este cel mai mare bit).
Dimensiunea maximă a unui vector poate fi limitată dar nu mai puțin de 2^{16} biți.
- Dimensiune_tablou este de forma [prima_adresă : ultima_adresă].
Cele două valori trebuie să fie întregi, parametri întregi sau o expresie care se rezumă la o constantă întreagă. Pot fi utilizate sau ordinea crescătoare sau cea descrescătoare.
Dimensiunea maximă a unui tablou poate fi limitată dar nu mai puțin de 2^{24} biți.
- Timp_decay (opțional) specifică volumul de timp în care o legătură **trireg** va memora o încărcare după ce toate sursele se vor opri, înainte de ajungerea la valoarea logică X.
Sintaxa este (întârziere_crescătoare, întârziere_căzătoare, timp_decay). Valoare de "default" a timpului este infinit.

Verilog are două tipuri primare de date:

- Legătură – reprezintă conexiuni structurale între componente
- Registru – reprezintă variabile utilizate pentru a memora date

Fiecare semnal are un tip de date asociat cu el, astfel:

- Prin declarare explicită: cu o declarație în codul **Verilog**
- Prin declarare implicită: fără declarare când se conectează structural blocurile constructive din cod; aceasta este întotdeauna o legătură de tip "wire" și de dimensiune un bit.

Tipuri legătură

Fiecare tip de legătură are o funcționalitate care este utilizată pentru a modela diferite tipuri de "hardware" (cum ar fi PMOS, NMOS, CMOS, etc).

Există următoarele tipuri de legături:

| Mnemonică | Funcționalitatea |
|-------------------------------|---|
| wire sau tri | Interconectare simplă de fire |
| wor sau trior | leșiri legate prin SAU cablat (model ECL) |
| wand sau triand | leșiri legate prin ȘI cablat (model "open-collector") |
| tri0 | Legătura la masă cu trei-stări ("Pulls down" cu „tri-stated") |
| tri1 | Legătura la sursă cu trei-stări ("Pulls up" cu „tri-stated") |
| supply0 | Constantă logică 0 ("supply strength") |
| supply1 | Constantă logică 1 ("supply strength") |
| triereg | Reține ultima valoare când trece în trei-stări ("capacitance strength") |

Tipul "wire" este cel mai folosit în proiectele **Verilog**.

Exemple:

```
wor a; sau wand a;
```

```
reg b, c;
```

```
assign a = b;
```

```
assign a = c; //a = b | c sau a = b & c si pleaca cu valoarea x
```

```
sau
```

```
tri a; sau triereg a;
```

```
reg b, c;
```

```
assign a = (b) ? c : 1'bz; // a pleaca cu valoarea z si va prelua c sau z
```

```
// sau a pleaca cu valoarea x si va prelua ultima
```

```
// valoare c
```

Legăturile transferă atât valori logice, cât și puteri (“strengths”) logice.

O legătură de date trebuie utilizată când:

- un semnal este dat de către ieșirea unui anumit dispozitiv
- un semnal este declarat ca port de intrare sau bidirecțional
- un semnal este de partea stângă a unei atribuiri continue

Tipuri registru

Registrele memorează ultima valoare atribuită lor până ce o altă atribuire le schimbă valoarea. Ele reprezintă constructori de memorare a datelor. Se pot crea rețele de registre numite memorii. Tipurile de date registru sunt utilizate ca variabile în blocurile procedurale. Un astfel de tip este cerut dacă un semnal este atribuit unei valori printr-un bloc procedural (aceste blocuri încep cu **initial** sau **always**).

Există următoarele tipuri de registre:

| Mnemonica | Funcționalitatea |
|---------------------------------|---|
| reg | Variabilă fără semn pe orice dimensiune de biți |
| integer | Variabilă cu semn pe 32 biți |
| time | Variabilă fără semn pe 64 biți |
| real sau realtime | Variabilă în virgulă mobilă cu dublă precizie |

Tipul “**reg**” este cel mai folosit în proiectele **Verilog**. Tipurile de date registru sunt utilizate ca variabile în blocurile procedurale. Ele memorează numai valori logice (nu “strengths”) și trebuie utilizate când semnalul este pe stânga unei atribuiri procedurale.

Șiruri

Un șir este o secvență de caractere încadrate prin ghilimele și toate conținute într-o singură linie. Șirurile utilizate ca operanzi în expresii și atribuiri sunt tratate ca o secvență de valoare ASCII pe 8 biți, valoare ce reprezintă un caracter. Pentru a declara o variabilă să memoreze un șir, se declară un registru suficient de mare pentru a ține numărul maxim de caractere pe care variabila îl păstrează; în **Verilog** nu se cer biți suplimentari pentru a păstra o terminație de caracter. Șirurile pot fi manipulate utilizând operatori standard. Când o variabilă este mai mare decât este cerută să păstreze o valoare în timpul atribuiri, **Verilog** umple conținutul la stânga cu zerouri după atribuire. Acest fapt asigură consistența în timpul atribuiri unor valori non-șir.

Anumite caractere pot fi utilizate în șiruri numai când sunt precedate de un caracter introductiv numit caracter de scăpare ("escape"). Aceste caractere speciale sunt următoarele:

| Caracter | Descriere |
|----------|--|
| \n | Caracter linie nouă |
| \t | Caracter "Tab" |
| \\ | Caracter "backslash" (\) |
| \" | Caracter ghilimele |
| \ddd | Caracter specificat în 1-3 digiți octali (0 <= d <= 7) |
| %% | Caracter procent (%) |

Exemplu:

```
module strings();
// Declara o variabila registru pe 29 octeti
reg [8*29:0] string ;
```

initial begin

```
    string = "Acesta este un exemplu de sir";
    $display ("%s \n", string);
```

end

endmodule

Alte tipuri de date

| Alte tipuri | Funcționalitate |
|------------------|---|
| parameter | Constantă "run-time" pentru memorarea întregilor, numerelor reale, timpului, întârzierilor sau șirurilor ASCII. Parametrii pot fi redefiniți pentru fiecare instanță a modului. |
| specparam | Specifică constanta bloc pentru memorarea întregilor, numerelor reale, timpului, întârzierilor sau șirurilor ASCII. |
| event | Un "flag" temporar fără valoare logică sau memorare date. Este utilizat adesea pentru sincronizarea activităților concurente dintr-un modul. |

2.3. OPERATORI DE LIMBAJ

Operatorii execută o operație pe unul sau doi operanzi:

- Expresii unare: *operator operand*
- Expresii binare: *operand operator operand*

Operanzii pot fi de tip legături sau registru și pot fi scalar, vector sau o selecție de biți ai unui vector. Operatorii care întorc un rezultat adevărat/fals vor întoarce o valoare pe 1 bit cu valoare 1 pentru adevărat, 0 pentru fals și x pentru nedeterminat.

2.3.1. OPERATORI ARITMETICI

| Simbol | Utilizare | Descriere |
|--------|-----------|---|
| + | $m + n$ | Adună n la m |
| - | $m - n$ | Scade n din m |
| - | $-m$ | Negatul lui m (în complement față de 2) |
| * | $m * n$ | Înmulțește m cu n |
| / | m / n | Împarte m la n |
| % | $m \% n$ | Modulul de m / n |

Împărțirea întregă trunchiază orice parte fracțională. Rezultatul operației *modul* ia semnul primului operand. Dacă oricare bit al unui operand are valoare "x", atunci întregul rezultat este "x". Tipurile de date registru sunt utilizate ca valori fără semn (numerele negative sunt memorate în cod complementar).

2.3.2. OPERATORI LA NIVEL DE BIT

| Simbol | Utilizare | Descriere |
|--------------------------------|--|--|
| ~ | $\sim m$ | Inversează (neagă) fiecare bit al lui m |
| & | $m \& n$ | ȘI logic fiecare bit al lui m cu fiecare bit al lui n |
| | $m n$ | SAU logic fiecare bit al lui m cu fiecare bit al lui n |
| ^ | $m \wedge n$ | SAU EXCLUSIV fiecare bit al lui m cu fiecare bit al lui n |
| $\sim \wedge$ $\wedge \sim$ | $m \sim \wedge n$ $m \wedge \sim n$ | SAU-NU EXCLUSIV fiecare bit al lui m cu fiecare bit al lui n |

Dacă un operand este mai scurt decât altul, el se va extinde la stânga cu zerouri până ce se potrivește ca lungime cu operandul mai lung. Operațiile consideră și biți de valoare "x".

2.3.3. OPERATORI DE REDUCERE

| Simbol | Utilizare | Descriere |
|----------|------------|--|
| & | &m | ȘI logic împreună pe toți biții din m (1 - Adevărat/1 - Fals) |
| ~& | ~&m | ȘI-NU logic împreună pe toți biții din m (1 - Adevărat/1 - Fals) |
| | m | SAU logic împreună pe toți biții din m (1 - Adevărat/1 - Fals) |
| ~ | ~ m | SAU-NU logic împreună pe toți biții din m (1 - Adevărat/1 - Fals) |
| ^ | ^m | SAU EXCLUSIV împreună pe toți biții din m (1 - Adevărat/1 - Fals) |
| ~^ ^^ | ~^m ^^m | SAU-NU EXCLUSIV împreună pe toți biții din m (1 - Adevărat/1 - Fals) |

Operatorii de reducere sunt unari și produc rezultat pe 1 bit. Operațiile de reducere consideră și biții de valoare "x".

2.3.4. OPERATORI LOGICI

| Simbol | Utilizare | Descriere |
|--------|-----------|---------------------------------------|
| ! | !m | m negat logic (1 - Adevărat/1 - Fals) |
| && | m && n | m ȘI logic n (1 - Adevărat/1 - Fals) |
| | m n | m SAU logic n (1 - Adevărat/1 - Fals) |

Expresiile conectate prin && sau || sunt evaluate de la stânga la dreapta. Rezultatul este "x" dacă oricare operand conține biți de valoare "x".

2.3.5. OPERATORI DE EGALITATE

| Simbol | Utilizare | Descriere |
|--------|-----------|---|
| == | m == n | Este m egal cu n? (1 - Adevărat/1 - Fals) |
| != | m != n | Este m neegal (diferit) cu (de) n? (1 - Adevărat/1 - Fals) |
| === | m === n | Este m identic cu n? (1 - Adevărat/1 - Fals) |
| !== | m !== n | Este m neidentic (diferit) cu (de) n? (1 - Adevărat/1 - Fals) |

Egalitatea este logică (rezultatul este "x" dacă operatorii conțin "x" sau "z") iar identitatea este cazuistică (include și comparații de "x" și "z").

Operanzii sunt comparați bit cu bit, cu adăugări de zerouri dacă cei doi operanzi nu au aceeași dimensiune.

2.3.6. OPERATORI RELAȚIONALI

| Simbol | Utilizare | Descriere |
|--------|------------|---|
| < | $m < n$ | Este m mai mic decât n? (1 - Adevărat/1 - Fals) |
| > | $m > n$ | Este m mai mare decât n? (1 - Adevărat/1 - Fals) |
| <= | $m \leq n$ | Este m mai mic sau egal decât n? (1 - Adevărat/1 - Fals) |
| >= | $m \geq n$ | Este m mai mare sau egal decât n? (1 - Adevărat/1 - Fals) |

Dacă oricare dintre operanzi este "x" sau "z", atunci rezultatul este considerat fals.

2.3.7. ALȚI OPERATORI

| Operatori deplasare logică | | |
|----------------------------|-----------|---------------------------------|
| Simbol | Utilizare | Descriere |
| << | $m \ll n$ | Deplasare m la stânga de n-ori |
| >> | $m \gg n$ | Deplasare m la dreapta de n-ori |

Pozițiile vacante sunt umplute cu zerouri.

| Miscellaneous Operators | | |
|-------------------------|--------------|---|
| Simbol | Utilizare | Descriere |
| ? : | $sel?m:n$ | Operator condițional: dacă sel este adevărat, selectează m; altfel selectează n |
| {} | $\{m,n\}$ | Concatenează m la n, realizând un vector mai mare |
| {{}} | $\{n\{m\}\}$ | Replică m de n-ori |
| -> | $-> m$ | Poziționează un semafor pe un tip de date event |

Concatenarea nu permite numere constante fără semn. Se pot utiliza multiplicatori (constante) de repetare (exemplu: `-> {5{b}}` // echivalent cu `{b, b, b, b, b}`).

Sunt posibile operații cu operatori de concatenare și replicare îmbricați (exemplu: `-> {a, {4{b, c}}}` // echivalent cu `{a, b, c, b, c, b, c, b, c}`).

2.3.8. PRECEDENȚA OPERATORILOR

| Precedența operatorilor | | |
|-------------------------|---------------------------|----------------------------|
| ! ~ + - (unari) | Cea mai înaltă precedență | |
| * / % | | |
| + - (binari) | | |
| << >> | | |
| < <= > >= | | |
| == != === !== | | |
| & ~& | | |
| ^ ~^ | | |
| ~ | | |
| && | | |
| | | |
| ?: | | Cea mai scăzută precedență |

Operatorii de pe aceeași linie au aceeași precedență și asociază termenul din stânga celui din dreapta într-o expresie.

Parantezele sunt utilizate pentru a modifica precedența sau pentru a clarifica situația.

Pentru a ușura înțelegerea, se recomandă utilizarea parantezelor.

2.4. CONSTRUCȚII/INSTRUCȚIUNI VERILOG

2.4.1. BLOCURI PROCEDURALE

De regulă codul comportamental **Verilog** există în interiorul blocurilor procedurale, dar există și excepții care vor fi prezentate ulterior.

În Verilog există două tipuri de blocuri procedurale:

- **initial**: se execută numai o dată, la momentul de timp zero (când se pornește execuția)
- **always**: aceste bucle se execută mereu și mereu, adică, după cum sugerează și numele, se execută întotdeauna până la oprirea simulării

Sintaxa blocurilor procedurale este următoarea:

| Sintaxa |
|---|
| <pre>tip_bloc @(listă_senzitivitate) grup_instrucțiuni: nume_grup declarații_variabibile_locale control_sincronizare instrucțiuni_procedurale sfârșit_grup_instrucțiuni</pre> |

unde:

- **tip_bloc**: este unul din cele două tipuri de blocuri procedurale
- **listă_senzitivitate**: este un eveniment de control al sincronizării care stabilește când vor începe a fi evaluate toate instrucțiunile din blocul procedural; lista este utilizată pentru a modela comportarea logică combinațională și secvențială
- structura **grup_instrucțiuni - sfârșit_grup_instrucțiuni**: este utilizată pentru a grupa împreună două sau mai multe instrucțiuni procedurale și controlează ordinea de execuție
 - **begin – end** grupează împreună două sau mai multe instrucțiuni în mod secvențial, astfel încât instrucțiunile să fie evaluate în ordinea în care sunt listate; fiecare control de sincronizare este relativ la instrucțiunea precedentă iar blocul se termină după ultima instrucțiune din bloc
 - **fork – join** grupează împreună două sau mai multe instrucțiuni în mod paralel, astfel încât toate instrucțiunile sunt evaluate concurrent; fiecare control de sincronizare este absolut la momentul când grupul începe să se execute iar blocul se termină când după ultima instrucțiune (instrucțiunile cu cea mai mare întârziere pot fi primele din bloc)

- Pot exista construcții **fork – join** în interiorul unei construcții **begin – end**
- `nume_grup` (opțional): crează un scop în grupul de instrucțiuni; grupurile (blocurile) denumite pot avea variabile locale și pot fi dezactivate prin construcția **disable**
- `declarații_variabibile_locale` (opțional): trebuie să fie un tip de date registru
- `control_sincronizare`: este utilizat pentru a controla atunci când se execută instrucțiunile dintr-un bloc procedural; va fi prezentat ulterior
- `instrucțiuni_procedurale`: sunt atribuiri procedurale la o variabilă registru (**reg, integer, time**) sau instrucțiuni program și vor fi prezentate ulterior; atribuiri procedurale pot atribui valori de tip legătură (**wire**), constante, un alt registru sau o valoare specific iar instrucțiunile program pot fi instrucțiuni condiționale, de caz, de repetare sau dezactivare

2.4.2. ATRIBUIRI PROCEDURALE

Pot fi atribuiri de tip blocante sau nonblocante.

Atribuiri blocante se execută în ordinea în care au fost scrise, adică sunt secvențiale. Ele au simbolul “=” . Aceste atribuiri acționează ca și în limbajele tradiționale de programare: instrucțiunea următoare nu se execută până ce întreaga instrucțiune curentă este complet executată și doar atunci se trece controlul execuției la instrucțiunea următoare celei curente.

Sintaxa unei atribuiri blocante este următoarea:

```
nume_tip_date_registru = expresie;
```

Atribuiri nonblocante sunt executate în paralel. Ele au simbolul “<=” . Instrucțiunea următoare nu este blocată în timpul execuției instrucțiunii curente. Atribuirea nonblocantă evaluează termenul din dreapta, pentru unitatea curentă de timp și atribuie valoarea obținută termenului din stânga, la sfârșitul unității de timp.

Sintaxa unei atribuiri nonblocante este următoarea:

```
nume_tip_date_registru <= expresie;
```

Exemplu:

```
//se presupune că inițial a = 1
// testarea atribuiri blocante
```

```

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

    end                   //rezultat final a=4
// testarea atribuirii nonblocante
always @(posedge clk)
    begin
        a <= a+1;         //în acest moment a=2
        a <= a+2;         //în acest moment a=3

    end                   //rezultat final a=3

```

Efectul este acela că, pentru toate atribuirile nonblocante se folosesc vechile valori ale variabilelor, de la începutul unității curente de timp, pentru a asigura registrelor noile valori, la sfârșitul unității curente de timp.

Aceasta reprezintă o reflectare a modului în care apar unele transferuri între registre, în unele sisteme hardware.

Controlul sincronizării va fi prezentat ulterior.

Există următoarele tipuri de atribuire procedurale, blocante sau nonblocante:

Assign și deassign

Aceste construcții de atribuire procedurală permit ca atribuirile continue să fie plasate pe registre pentru controlul perioadelor de timp. Construcția **assign** se extinde peste atribuirile procedurale la o variabilă registru. Construcția **deassign** dezactivează o atribuire continuă la o variabilă registru.

Sintaxa acestora este următoarea:

assign nume_tip_date_registru = expresie;

deassign nume_tip_date_registru;

Force și release

Aceste construcții au efect similar ca al perechii „assign – deassign” dar pot fi aplicate și legăturilor (“wire”), nu numai registrelor; forțează atribuirea și deatribuirea oricăror tipuri de date la o valoare.

Se pot utiliza în timpul simulării la nivel de porți în cazul problemelor de conectivitate la “reset”.

De asemenea, pot fi utilizate pentru a insera unul sau doi biți de eroare la o citire de date de la memorie.

Sintaxa acestora este următoarea:

```
force nume_tip_date_registru sau legătură = expresie;
```

```
release nume_tip_date_registru sau legătură;
```

2.4.3. CONSTRUCȚII/INSTRUCȚIUNI CONDIȚIONALE

Construcția **if – else** controlează execuția altor instrucțiuni și poate include sincronizări procedurale.

Când mai mult de o instrucțiune necesită a fi executată pentru o condiție **if**, atunci se vor utiliza **begin** și **end** în interiorul construcției.

Sintaxa instrucțiunilor condiționale este următoarea:

| Sintaxa |
|---|
| <p style="text-align: center;">Instrucțiune if</p> <pre>if (expresie_condiție) // dacă expresia evaluată este adevărată instrucțiune sau grup de instrucțiuni // execută instrucțiunea sau grupul de instrucțiuni</pre> |
| <p style="text-align: center;">Instrucțiune if – else</p> <pre>if (expresie_condiție) // dacă expresia evaluată este adevărată instrucțiune sau grup de instrucțiuni // execută instrucțiunea sau grupul de instrucțiuni else // altfel dacă expresia evaluată este falsă sau necunoscută instrucțiune sau grup de instrucțiuni // execută instrucțiunea sau grupul de instrucțiuni</pre> |
| <p style="text-align: center;">Instrucțiune îmbricată if – else – if</p> <pre>if (expresie1_condiție) // dacă expresia evaluată este adevărată instrucțiune sau grup de instrucțiuni // execută instrucțiunea sau grupul de instrucțiuni else if (expresie2_condiție) // altfel dacă expresia1 evaluată este falsă sau necunoscută //și dacă expresia2 evaluată este adevărată instrucțiune sau grup de instrucțiuni // execută instrucțiunea sau grupul de instrucțiuni</pre> |

În mod normal nu se include verificarea de **reset** în *expresia_condiție*. Așa că atunci când este necesară o logică de priorități, se utilizează construcția **if – else – if**.

Pe de altă parte dacă nu se dorește implementarea unei logici de priorități, știind că numai o intrare este activă la un moment dat (adică toate intrările sunt mutual exclusive), atunci se poate scrie codul **Verilog** al logicii într-un **if** paralel de forma **if – else – if – if**.

2.4.4. CONSTRUCȚII/INSTRUCȚIUNI DE CAZ

Construcțiile de caz compară o expresie cu o serie de cazuri și execută instrucțiunea sau grupul de instrucțiuni asociat cu prima potrivire de caz. În cazul grupului de instrucțiuni multiple se utilizează **begin** – **end**.

Sintaxa instrucțiunilor de caz este următoarea:

| Sintaxa | |
|---|---|
| Instrucțiune case | |
| case (legătură_sau_registru_sau_simbol literal) | // compară legătura, registrul sau |
| potrivire_caz1: instrucțiune sau grup de instrucțiuni | // valoarea simbolului literal cu fiecare caz și |
| potrivire_caz2, | // execută instrucțiunea sau grupul de instrucțiuni |
| potrivire_caz3: instrucțiune sau grup de instrucțiuni | // asociat cu prima potrivire de caz |
| default: instrucțiune sau grup de instrucțiuni | // execută predefinit (opțional) dacă |
| endcase | // nici un caz nu se potrivește |
| Instrucțiune casez | |
| casez (legătură_sau_registru_sau_simbol literal) | // tratează z ca “nu contează” |
| Instrucțiune casex | |
| casex (legătură_sau_registru_sau_simbol literal) | // tratează x și z ca “nu contează” |

Când nu se folosește **default** atunci trebuie precizate toate cazurile.

Dacă la mai multe cazuri se execută aceeași instrucțiune sau grup de instrucțiuni atunci se pot specifica cazurile multiple ca un singur caz, după cum se prezintă în exemplul de mai jos:

```
module mux_fara_default (a,b,c,d,sel,y);
input a, b, c, d;
input [1:0] sel;
output y;

reg y;

always @ (a or b or c or d or sel)
case (sel)
0 : y = a;
1 : y = b;
2 : y = c;
3 : y = d;
2'apox;bxx,2'bx0,2'bx1,2'b0x,2'apox;b1x,
2'apox;bzz,2'bz0,2'bz1,2'b0z,2'apox;b1z : $display("Eroare de selectie");
endcase

endmodule
```

2.4.5. CONSTRUCȚII/INSTRUCȚIUNI PENTRU REPETARE

Construcțiile de repetare apar numai în interiorul blocurilor procedurale. **Verilog** are patru astfel de construcții (**forever**, **repeat**, **while**, **for**) și vor fi prezentate în continuare.

Forever

Bucloa **forever** se execută continuu, ea nu se termină niciodată. În mod normal se utilizează această instrucțiune în blocurile **initial**.

Sintaxa este următoarea:

forever *instrucțiune sau grup de instrucțiuni*

Trebuie acordată o atenție deosebită la utilizarea acestei instrucțiuni: simularea se poate agăța dacă nu este prezent un constructor de sincronizare.

Repeat

Bucloa **repeat** se execută de un număr fixat de ori. Numărul poate fi un întreg, o variabilă sau o expresie (o variabilă sau o expresie este evaluată numai când se intră prima dată în buclă).

Sintaxa este următoarea:

repeat (număr) *instrucțiune sau grup de instrucțiuni*

Trebuie la fel avută grijă pentru sincronizarea instrucțiunilor multiple din interiorul buclei.

While

Bucloa **while** se execută atât timp cât expresia este evaluată ca adevărată.

Sintaxa este următoarea:

while (expresie) *instrucțiune sau grup de instrucțiuni*

For

Sintaxa este următoarea:

for (*atribuire_inițială; expresie; atribuire_pas*) *instrucțiune sau grup de instrucțiuni*

Bucloa **for**, la fel ca în orice limbaj de programare:

- Execută o atribuire inițială o dată la startul buclei
- Execută bucla atât timp cât expresia este evaluată ca adevărată
- Execută o atribuire de pas la sfârșitul fiecărei treceri prin buclă

Exemplu:

```
...
for (i = 0; i < 256; i = i + 1) begin
...

```

Trebuie la fel avută grijă pentru sincronizarea instrucțiunilor multiple din interiorul buclei.

2.4.6. CONSTRUCȚII/INSTRUCȚIUNI DE DEZACTIVARE

Sintaxa este următoarea:

```
disable nume_grup;
```

Înterupe execuția unui grup denumit de instrucțiuni. Simularea acestui grup sare la sfârșitul grupului fără execuția oricărui eveniment programat.

2.4.7. ATRIBUIREA CONTINUĂ; PROPAGAREA ÎNTÂRZIERILOR

Construcțiile de atribuirii continue manipulează legături (tipuri de date fire). Ele reprezintă conexiuni structurale.

Sintaxa este următoarea:

| Sintaxa |
|--|
| <p style="text-align: center;">Atribuire continuă explicită</p> <pre>tip_legătură [<i>dimensiune</i>] <i>nume_legătură</i>; assign ("strength") #(întârziere) <i>nume_legătură</i> = <i>expresie</i>;</pre> |
| <p style="text-align: center;">Atribuire continuă implicită</p> <pre>tip_legătură [<i>dimensiune</i>] ("strength") <i>nume_legătură</i> = <i>expresie</i>;</pre> |

- Atribuirile continue explicite cer două instrucțiuni: una pentru a declara legătura și una pentru a atribui continuu o valoare la aceasta.
- Atribuirile continue implicite combină declararea legăturii și atribuirea continuă într-o singură instrucțiune.

- Tip_legătură poate fi orice tip de date legătură cu excepția **trireg**.
- “Strength” (opțional) poate fi specificat numai când atribuirea continuă este combinată cu o declarație de legătură. Valoarea predefinită pentru “strength” este (strong1, strong0).
- Întârzierea (opțional) urmărește aceeași sintaxă ca la instanțierea primitivelor. Valoarea predefinită este zero.
- Expresia poate include orice tip de date, orice operator și apelări de funcții.
- Atribuirile continue modelează tamponare (“buffers”) cu impedanță înaltă și pot modela logica de tip combinațional. De fiecare dată când un semnal se schimbă pe partea din dreapta, partea dreaptă este reevaluată și rezultatul este atribuit legăturii din partea stângă.
- Atribuirile continue sunt declarate în afara blocurilor procedurale și se extind peste orice atribuire procedurală. Ele devin automat active la momentul zero și sunt evaluate concurrent cu blocurile procedurale, instanțierile de module și de primitive.

Exemple:

```

...
reg a, b;
wire suma, transport_out;

assign #5 { transport_out,suma} = a+b;
...sau

...
wire [0:31] (strong1, pull0) ual_out = functie_ual(cod_op,a,b);
...sau

...
reg date_in, activare;
wire date_out;

assign date_out = (activare) ? date_in : 1'bz;
...sau

...
tri [0:15] #5 date_out = (activare) ? date_in : 16'bz;
...etc.

```

Propagarea întârzierilor

Atribuirile continue pot avea o întârziere specificată și numai pentru toate tranzițiile. Poate fi specificat și un domeniu de întârzieri minim:tipic:maxim.

Exemplu:

```
assign #(1:2:3) date_out = (activare) ? date_in : 1'bz;
```

2.4.8. CONTROLUL BLOCURILOR PROCEDURALE; BLOCURI NUMITE

Blocurile procedurale devin active la momentul zero de simulare. Pentru a controla execuția unei proceduri se utilizează evenimente senzitive pe nivel. Nu se poate controla un bloc cu o variabilă căreia blocul îi atribuie valori sau pe care el le manipulează. Dacă în interiorul unui modul există blocuri **always** multiple, atunci toate blocurile (**always** și **initial**) vor începe să se execute de la momentul zero și vor continua să se execute concurrent; deseori acest lucru poate conduce la condiții neprecizate și ieșirile nu pot fi stabilite.

Controlul procedural al logicii combinaționale

Pentru a modela logica de tip combinațional, un bloc procedural trebuie să fie sensibil la orice schimbare a intrărilor. Dacă se utilizează instrucțiunea condițională atunci este nevoie să se utilizeze și partea “else”; lipsa acestei părți schimbă logica într-un “latch”. Dacă nu se dorește utilizarea “else” atunci trebuie să se inițializeze toate variabilele din blocul combinațional chiar de la început.

Controlul procedural al logicii secvențiale

Pentru a modela logica de tip combinațional, un bloc procedural trebuie să fie sensibil la frontul pozitiv sau negativ al semnalului de ceas. Dacă se dorește o inițializare asincronă atunci blocul trebuie să fie sensibil și la frontul semnalelor de “reset”, “preset”, “clear”, etc. Toate atribuirile la logica secvențială trebuie să fie realizate prin atribuiri procedurale nonblocante. La simulare se pot utiliza fronturi multiple ale variabilelor dar la sinteză numai un front al variabilei contează în funcționarea reală; nu este bine la sinteză să se utilizeze semnalul de ceas pentru a activa bistabile.

Blocuri numite

Blocurile pot fi numite prin adăugarea construcției “: nume_bloc” după **begin**. Blocurile numite pot fi dezactivate prin utilizarea instrucțiunii “disable” în interiorul blocului.

2.4.9. BLOCURI SPECIFICE

Au următoarea sintaxă:

| Sintaxa |
|---|
| specify declarații specparam verificări_constrângerî_timing întârziere_cale_simplă_pin-la-pin întârziere_cale_front-senzitiv_pin-la-pin întârziere_cale_stare-dependentă_pin-la-pin endspecify |

Declarații specparam

Au sintaxa:

specparam nume_parametru = valoare, nume_parametru = valoare, ...;

Parametrii specifici sunt constant utilizate pentru a memora întârzieri, factori de calcul ai întârzierii, factori de sinteză, etc. Valorile lor pot fi întregi, reali sau șiruri.

Verificări constrângerî de "timing"

Sunt task-uri sistem care modelează restricțiile la schimbările pe intrări, cum sunt timpîi de "setup" și de "hold".

Au sintaxa următoare:

| Sintaxa |
|--|
| \$setup (eveniment_date, eveniment_referință, limită_setup, notificare); |
| \$hold (eveniment_date, eveniment_referință, limită_hold, notificare); |
| \$setuphold (eveniment_referință, eveniment_date, limită_setup, limită_hold, notificare); |
| \$skew (eveniment_referință, eveniment_date, limită_skew, notificare); |
| \$recovery (eveniment_referință, eveniment_date, limită, notificare); |
| \$period (eveniment_referință, limită_period, notificare); |
| \$width (eveniment_referință, width_limită, width_threshold, notificare); |

Caracteristici:

- Verificările de “timing” pot fi utilizate numai în blocuri specifice.
- Evenimentul de referință este un front al unui semnal de intrare care stabilește un punct de referință pentru schimbările unui eveniment de date. Evenimentul de date este semnalul de intrare care este monitorizat pentru schimbări. Evenimentele de date și de referință trebuie să fie porturi de intrare.
- Limită și “threshold” sunt valori de întârzieri și utilizează aceeași sintaxă la întârzierile singulare de primitive.
- Notificarea (opțional) este o variabilă registru utilizată ca un indicator. Când apare o violare de “timing”, funcționalitatea modelului poate utiliza indicatorul de notificare pentru a modifica ieșirile modelului.

Întârzieri cale pin-la-pin

- Întârziere cale simplă:
(*port_intrare* polaritate:indicator_cale *port_ieșire*) = (întârziere);
- Întârziere cale senzitivă pe front:
(front *port_intrare* indicator_cale (*port_ieșire* polaritate:sursă)) = (întârziere);
 - Front (opțional) poate fi *posedge* sau *negedge*. Dacă nu este specificat sunt utilizate toate tranzițiile de intrare.
 - Sursa (opțional) este portul de intrare sau valoarea de ieșire care va fi primită. Sursa este ignorată de către majoritatea simulatoarelor logice, dar poate fi utilizată de către analizoarele de “timing”.
- Întârziere cale dependentă de stare:
if (prima_condiție) *întârziere_cale_simplă_sau_front-senzitiv*
if (următoarea_condiție) *întârziere_cale_simplă_sau_front-senzitiv*
ifnone *întârziere_cale_simplă*
 - Permite întârzieri diferite pentru aceeași cale care urmează a fi specificată, pe baza altor condiții de intrare.
 - Condiția poate fi bazată numai pe porturi de intrare.

- Cu condiții pot fi utilizați majoritatea operatorilor, dar trebuie să se rezume la adevărat/fals (o valoare logică "x" sau "z" este considerată adevărată; dacă o condiție se rezumă la un vector este utilizat numai *lsb*).
 - Fiecare întârziere dependentă de stare pentru aceeași cale trebuie să aibă o condiție diferită sau un front-senzitiv diferit.
 - Condiția **ifnone** (opțional) poate fi numai o întârziere de cale simplă și servește ca valoare predefinită dacă nu este evaluată nici o condiție ca adevărată.
- Polaritatea (opțional) este ori + sau P și indică dacă intrarea va fi sau nu va fi inversată. Indicatorul de polaritate este ignorat de către majoritatea simulatoarelor logice, dar poate fi utilizată de către analizoarele de "timing".
 - Indicatorul de cale este ori simbolul *> pentru conexiune completă sau simbolul => pentru conexiune paralelă. Întârzierea de cale complet conectată indică faptul că fiecare bit de intrare poate avea o cale de întârziere la fiecare bit de ieșire. Întârzierea de cale conectată paralel indică faptul că fiecare bit de intrare este conectat la bitul său corespondent de ieșire (bitul 0 la bitul 0, bitul 1 la bitul 1, ...).
 - Întârzierea reprezintă faptul că pot fi specificate 1, 2, 3, 6 sau 12 tranziții. Fiecare tranziție poate avea o întârziere singulară sau un domeniu de întârziere minim:tipic:maxim.

| Întârzieri | Tranziții reprezentate (în ordine) |
|------------|---|
| 1 | Toate tranzițiile de ieșire |
| 2 | Tranzițiile de ieșire crescătoare și descrescătoare |
| 3 | Tranzițiile de ieșire crescătoare, descrescătoare și "turn-off" |
| 6 | Tranziții crescătoare, descrescătoare, 0->Z, Z->1, 1->Z, Z->0 |
| 12 | Tranziții crescătoare, descrescătoare, 0->Z, Z->1, 1->Z, Z->0, 0->X, X->1, 1->X, X->0, X->Z, Z->X |

2.4.10. CONTROLUL SINCRONIZĂRII ÎN BLOCURILE PROCEDURALE

Controlul sincronizării se poate realiza prin controlul întârzierii, prin controlul evenimentelor sensitive pe front sau prin controlul evenimentelor sensitive pe nivel.

Controlul întârzierii

Sintaxa este următoarea: *#întârziere*

Întârzie execuția unei construcții procedurale pentru o valoare specifică de timp. Întârzierea poate fi orice număr literal, o variabilă sau o expresie.

Controlul evenimentelor senzitive pe front

Are sintaxa: *@ (front_semnal sau front_semnal sau ...) instrucțiune;*

Întârzie execuția până ce apare o tranziție logică pe un semnal. Frontul (opțional) poate fi *posedge* sau *negedge*; dacă nu se specifică nici un front atunci este utilizată orice tranziție logică. Semnalul poate fi scalar sau vector și de orice tip de date.

Controlul evenimentelor senzitive pe nivel – construcții wait

Sintaxa este următoarea: *wait (expresie) #întârziere instrucțiune;*

Întârzie execuția până ce expresia este evaluată ca adevărată.

Alte sincronizări

Sintaxe posibile:

control_întârziere nume_tip_date_registru = expresie;

control_întârziere nume_tip_date_registru <= expresie;

Întârzie atribuirii procedurale. Evaluarea atribuirii este întârziată de un tip de control al sincronizării din cadrul celor prezentate mai sus.

nume_tip_date_registru = control_întârziere expresie;

Sunt atribuirii blocante întârziate intern. Expresia este evaluată în pasul de timp în care este întâlnită instrucțiunea și este atribuită într-o ordine nedeterministă în pasul de timp specificat prin controlul sincronizărilor anterioare.

nume_tip_date_registru <= control_întârziere expresie;

Sunt atribuiri nonblocante întârziate intern. Expresia este evaluată în fiecare pas de timp în care este întâlnită instrucțiunea și este atribuită la sfârșitul pasului de timp specificat prin controlul sincronizărilor anterioare. Sincronizarea modelează transportul întâzierii.

Controlul intern al atribuirilor evaluează întotdeauna expresia din partea dreaptă și atribuie rezultatul după întârziere sau eveniment de control

Ori de câte ori un semnal se schimbă în partea dreaptă, întreaga parte dreaptă este reevaluată și rezultatul este atribuit în partea stângă.

2.5. TASK-URI ȘI FUNCȚII

2.5.1. TASK-URI

Task-urile sunt utilizate ca în toate limbajele de programare și sunt cunoscute în general ca proceduri sau subrutine. Liniile de cod sunt delimitate de cuvintele cheie **task**, la început și **endtask**, la sfârșit.

Datele sunt trimise task_ului care le procesează și rezultatul este returnat la terminarea task_ului.

Task_urile sunt apelate în mod specific, cu date de intrare și de ieșire și sunt incluse codul programului **Verilog**. Ele pot fi apelate de multe ori, evitând repetarea codului.

Task_urile pot avea orice număr de intrări, ieșiri sau "inout"-uri și pot conține sincronizări de tip **#**, **@** sau **wait**.

Task-urile sunt definite în modulul în care sunt utilizate. Este posibil să se definească un task într-un fișier separat și să se utilizeze directiva de compilare "include" pentru a include task-ul în fișierul care instanțiază task-ul.

Sintaxa de definire a unui task este următoarea:

| Sintaxa |
|---|
| task nume_task; declarații <i>input</i> , <i>output</i> , și <i>inout</i> declarații variabile locale construcții procedurale sau grup_instrucțiuni endtask |

Caracteristici:

- Variabilele locale declarate într-un task sunt locale aceluși task.
- Ordinea de declarare dintr-un task definește cum sunt utilizate variabilele trimise task-ului de către apelanți.
- Când nu sunt utilizate variabile locale, task-urile pot manipula variabile globale (în acest caz variabilele sunt declarate ca registre înainte de definirea task-ului, în modulul în care se definește task-ul). Când se utilizează variabile locale, ieșirea este asignată unei variabile regidtru numai la sfârșitul execuției task-ului.
- Task-urile pot apela alte task-uri sau funcții.
- Task-urile pot fi utilizate pentru a modela atât logica combinațională, cât și logica secvențială.
- Un task trebuie să fie apelat specific cu o instrucțiune. El nu poate fi utilizat cu o expresie așa cum poate fi utilizată o funcție.

Exemplu:

Fie un task definit în fișierul **task-ul_meu.v**:

```
module definire_task;

task calculeaza; // definire task
input [0:15] a_in, b_in;
output [0:15] suma_out;
begin
    suma_out = a_in + b_in;
end
endtask

endmodule
```

Apelarea task-ului se realizează astfel:

```
module apelare_task (a_in, b_in, c_in, suma_out);
input [0:7] a_in, b_in, c_in;
output [0:7] suma_out;
reg [0:7] suma_out;
`include "task-ul_meu.v"

always @ (c_in)
begin
    calculeaza (a_in, b_in, suma_out); // apelare task
end

endmodule
```

2.5.2. FUNCȚII

O funcție **Verilog** este asemănătoare cu un task, cu foarte puține diferențe, cum ar fi faptul că funcția nu poate decât o ieșire și nu poate conține întârzieri. Funcțiile reîntorc ca ieșire valoarea care este atribuită numelui funcției. O funcție începe cu cuvântul cheie **function** și se termină cu cuvântul cheie **endfunction**.

Funcțiile sunt definite în modulul în care sunt utilizate. Este posibil să se definească funcții în fișiere separate și să se utilizeze directiva de compilare "include" pentru a include funcția în fișierul care instanțiază funcția.

Sintaxa de definire a unei funcții este următoarea:

| Sintaxa |
|---|
| <pre>function [dimensiune_sau_tip] <i>nume_funcție</i>; <i>declarații input</i> <i>declarații variabile locale</i> <i>construcții_procedurale sau grup_instrucțiuni</i> endfunction</pre> |

Caracteristici:

- Funcțiile trebuie să aibă cel puțin o intrare și nu pot avea ieșiri sau "inout"-uri.
- Ele nu pot conține sincronizări de tip **posedge**, **negedge**, **#**, **@** sau **wait**, ceea ce înseamnă că funcțiile se execută cu întârzierea zero.
- **Dimensiune_sau_tip** (opțional) este domeniul de biți reîntors ca *[msb : lsb]*, sau cuvântul cheie **integer** sau **real**.
- Variabilele declarate într-o funcție sunt locale acelei funcții. Ordinea de declarare dintr-o funcție definește cum sunt utilizate variabilele trimise funcției de către apelanți.
- Când nu sunt utilizate variabile locale, funcțiile pot manipula variabile globale (în acest caz variabilele sunt declarate înainte de definirea funcției, în modulul în care se definește funcția). Când se utilizează variabile locale, ieșirea este asignată unei variabile fir ("**wire**") numai la sfârșitul execuției funcției.
- Funcțiile pot apela alte funcții, nu însă și task-uri.
- Funcțiile pot fi utilizate pentru a modela doar logica combinațională.

Exemplu:

Fie o funcție definită în fișierul **functia_mea.v**:

```
module definire_functie;

function [15:0] functia_mea; // definire functie
input [7:0] a_in, b_in;
begin
    functia_mea = a_in * b_in;
end
endfunction

endmodule
```

Apelarea task-ului se realizează astfel:

```
module apelare_functie (a_in, b_in, c_in, produs);
input [7:0] a_in, b_in;
input c_in;
output [15:0] produs;
wire [7:0] produs;
`include "functia_mea.v"

assign produs = (c_in)? functia_mea (a_in, b_in) :16'b0; // apelare functie

endmodule
```

2.6. MODELAREA MEMORIILOR ȘI A MAȘINILOR ALGORITMICE DE STARE

Modelarea memoriilor

Pentru a ajuta la modelarea memoriilor, **Verilog** furnizează suport pentru tablouri cu două dimensiuni. Modelele comportamentale ale memoriilor sunt modelate prin declararea unui tablou de variabile registru; orice cuvânt din tablou poate fi accesat utilizând un index în tablou. Este cerută o variabilă temporară pentru a accesa un bit din cadrul tabloului.

Sintaxa este următoarea:

```
reg [msb_reg:0] nume_tablou [0:msb_tablou]
```

Exemplu:

```
reg [7:0] memoria_mea [0:1023] // memoria conține 1024 locații (adrese) de cuvinte de 8 biți,
// în ordine "little endian"
```

Scrierea se face prin atribuirea "*memoria_mea[adresă]= reg_in;*" iar citirea prin "*reg_out = memoria_mea[adresă];*".

Citirea unui bit se face după citirea cuvântului prin `“reg_out_bit_0 = reg_out[0];”`.

- **Inițializarea memoriilor**

Un tablou de memorie poate fi inițializat prin citirea fișierului “pattern”-ului de memorie de pe disk și memorarea acestuia în tabloul de memorie. Sintaxa este următoarea:

```
$readmemh_readmemb(“nume_fișier.extensie”,nume_tablou_memorie,adresă_start,adresă_stop);
```

Adresele și extensia fișierului sunt opționale.

Exemplu de fișier:

```
//Sunt permise comentarii si “underscore” intre digiti sau octeti, ...
1100_1100 // Aceast este prima adresa, 8'h00
1010_1010 // Aceasta este a doua adresa, 8'h01
@ 99 // Salt la noua adresa 8'h99
0101_1010 // Aceasta este adresa 8'h99
0110_1001 // Aceasta este adresa 8'h9A
...
```

Task-ul sistem `$readmemh` poate fi de asemenea utilizat pentru citirea vectorilor de test, după cum se va arăta în secțiunea programelor de test.

Modelarea mașinilor algoritmice de stare (FSM – “Flow State Machine”)

Diagramele de stări sau FSM sunt inima oricărui proiect digital și sunt necesare ori de câte ori avem și logică secvențială (cum este nevoie în majoritatea proiectelor **Verilog**).

Există două tipuri de diagrame: Moore, unde ieșirile sunt funcție numai de starea curentă și Mealy, unde una sau mai multe ieșiri sunt funcție de starea curentă și una sau mai multe intrări.

Diagramele de stări pot fi de asemenea clasificate după modul de codificare al stărilor: binar, complet decodificată, gray, etc. Codificarea este un factor critic care decide viteza și complexitatea porțiilor pentru diagramele de stări.

Este bine de ținut minte un lucru important când se codifică FSM și anume că logica de tip combinațional și cea de tip secvențial trebuie să se găsească în două blocuri **always** diferite. De asemenea, utilizarea declarărilor de tip **parameter** sau **define** pentru a defini stările din FSM face codul mai ușor de citit și de gestionat.

Codul **Verilog** trebuie să aibă trei secțiuni:

- Stilul codificării: Cel mai utilizat este binar și complet decodificat.
- Partea combinațională: Această secțiune poate fi modelată utilizând funcții, construcții de atribuire sau blocuri **always** cu o construcție **case**.
- Partea secvențială: Această secțiune trebuie să fie modelată utilizând logica senzitivă pe front cum sunt blocurile **always** cu *posedge* sau *negedge* de semnal de ceas.

2.7. SCRIEREA PROGRAMELOR DE TEST

Scrierea unui program de test este la fel de complex ca și scrierea codului RTL însuși. În zilele actuale circuitele ASIC devin din ce în ce mai complexe și astfel verificarea acestora devine o provocare. În mod tipic 60-70% din timpul necesar pentru proiectarea unui ASIC este cheltuit pe partea de verificare/validare/testare.

Pentru început, pentru scrierea programelor de test este important să existe specificațiile de proiectare ale “design under test” (DUT – proiectare în vederea testării). Specificațiile trebuie înțelese clar și este necesar să fie realizat în detaliu un plan de test cu o documentare temeinică a arhitecturii programului de test și a scenariilor de test. Primul pas în scrierea unui program de test este construirea unui “template” al lui, care va conține registre (**reg**) pentru intrări, fire (**wire**) pentru ieșiri apoi instanțiază proiectul.

Structura unui “template” este următoarea:

```
module nume_testbench;  
  reg [msb_r:0] intrări;  
  wire [msb_w:0] ieșiri;  
  
  nume_modul instanță (  
    .semnal1_modul (semnal1_instanță),  
    .semnal2_modul (semnal2_instanță),  
    ...  
  );
```

endmodule

Următorul pas este să se adauge logica de ceas. Înainte însă trebuie să se inițializeze diagrama la o stare cunoscută; acest lucru se realizează printr-un bloc **initial** care setează semnalele de intrare la valorile corespunzătoare primei stări care va fi setate. Logica de ceas se poate furniza prin mai multe moduri: se poate utiliza un bloc “**always #întârziere semnal_ceas = !semnal_ceas;**” sau se poate utiliza o buclă

forever într-un bloc **initial**. Se poate adăuga un *parameter* sau se poate utiliza *define* pentru a defini și controla întârzierea.

Acum se poate testa dacă programul de test generează semnalul de ceas în mod corect adăugând în programul de test în linia de comandă opțiuni de compilare și simulare, incluse în blocuri **initial**.

Astfel de opțiuni ar fi:

- *\$dumpfile*: este utilizat pentru specificarea fișierului pe care îl utilizează simulatorul pentru a memora formele de undă.
- *\$dumpvars*: instruieste compiliatorul **Verilog** să înceapă cu semnalele dintr-un fișier.
- *\$display*: este utilizat pentru a tipări pe ecran texte sau variabile.
- *\$monitor*: păstrează urma modificărilor variabilelor din listă; cum apare o schimbare ea va fi afișată în format specific.
- *\$finish*: este utilizat pentru terminarea simulării după un număr de unități de timp
- *\$readmemh*: este utilizat pentru citirea vectorilor de test, etc.

Logica de reset se poate furniza în multe moduri. Un prim mod ar fi printr-un bloc **initial** și apoi specificarea tranzițiilor printr-un bloc **always**, la fel ca pentru toate semnalele de intrare în proiectul **Verilog** de testat. Un alt mod, mai elegant, ar fi utilizarea unui **event** eşantionat, cum ar fi “reset_trigger”, la apariția căruia logica de reset va activa semnalul de reset pe frontul negativ de ceas și îl va dezactiva pe următorul front negativ de ceas; după dezactivare, logica de rest va eşantiona un alt **event** de sincronizare “reset_trigger_gata”.

Mergând mai departe, se poate adăuga logică pentru a genera cazuri de test. Astfel de cazuri pot fi:

- Reset test: se pornește cu *reset* dezactivat, apoi se activează *reset* pentru câteva perioade de ceas și apoi se dezactivează
- Enable test: activare/dezactivare *enable* după aplicarea *reset*
- Activare/dezactivare aleatoare pentru *enable* și *reset*

Cazurile de test nu pot exista în același fișier, așa că ele se codifică separat și sunt incluse apoi în programul de test utilizând directiva **include**. Se poate condiționa terminarea simulării de un **event** “terminate_sim”.

Se poate adăuga acum o logică de comparare a ceea ce se așteaptă să se obțină la simulare și ceea ce se obține de fapt. Această logică va face programul de test să se autotesteze automat.

2.8. CONCLUZII

Este bine să se adopte un stil de codificare al programelor **Verilog**. Acest stil presupune următoarele:

- Utilizarea unor nume sugestive pentru semnale și variabile
- A nu se mixa elemente senzitive pe nivel și pe front în același bloc
- A se evita mixarea de bistabile eșantionate pe front pozitiv și pe front negativ
- Utilizarea parantezelor în vederea optimizării structurii logice
- Utilizarea instrucțiunilor de atribuire continuă pentru logica simplă combinațională
- Utilizarea atribuirilor nonblocante pentru logica secvențială și a celor blocante pentru logice de tip combinațional
- A nu se mixa atribuiri blocante și nonblocante în același bloc **always**
- Atenție la atribuiri multiple la aceeași variabilă
- A se defini în mod explicit construcțiile **if-else** și **case**

În sinteza logică nu sunt suportați toți constructorii. Aceștia sunt următorii:

| Tip constructor | Observații |
|---------------------------|--|
| initial | Utilizat numai în programele de test |
| events | Utilizat numai în programele de test |
| real | Nesuportat |
| time | Nesuportat |
| force și release | Nesuportat |
| assign și deassign | Nesuportat pentru tipul de date registru dar suportat pentru tipul de date wire |
| fork și join | Nesuportat; a se utiliza atribuiri nonblocante în vederea aceluiași efect |
| primitive simple | Sunt suportate numai primitivele tip poartă |
| UDP | UDP și tabelele nu sunt suportate |

Sinteza logica nu suportă nici operatorii **===** și **!==**.

Furnizorii de tool-uri și utilizatorii de tool-uri pot defini task-uri și funcții specifice tool-ului lor, cum ar fi ieșiri de text sau afișare forme de undă. Task-urile sistem și funcțiile încep cu simbolul "\$".

Utilizatorii pot defini task-uri și funcții predefinite (“built-in”) utilizând interfața **Verilog** PLI (“Programming Language Interface”).

Pentru a controla cum vor interpreta modelele **Verilog** se furnizează directive de compilare. Acestea încep cu simbolul “`”

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. Dr. Hyde, C., Daniel, *Handbook on Verilog HDL*, Computer Science Department, Bucknell University, Lewisburg, PA 17837, August 25, 1995, Updated August 23, 1997.
4. Sternheim, E. , R. Singh, Y. Trivedi, R. Madhavan and W. Stapleton, *Digital Design and Synthesis with Verilog HDL*, published by Automata Publishing Co., Cupertino, CA, 1993, ISBN 0-9627488-2-X, \$65.
5. 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.
6. Bhasker, J., *A Verilog HDL Primer*, Star Galaxy Press, 1058 Treeline Drive, Allentown, PA18103, 1997, ISBN 0-9656277-4-8, \$60.
7. World Wide Web Pages:
Sutherland HDL, Inc. - http://www.sutherland-hdl.com/on-line_ref_guide/vlog_ref_top.html
ALDEC, Inc. - <http://www.aldec.com/products/tutorials/>
AWC, Inc. - <http://tutor.al-williams.com/wpv-1.htm>
Cadence Design Systems, Inc. - <http://www.cadence.com/>
Synopsis, Inc. – <http://www.synopsis.com>
Xilinx, Inc. – <http://www.xilinx.com>
Department of Electrical Engineering in the University of Edinburgh, Scotland, UK - <http://www.see.ed.ac.uk/~gerard/Teach/Verilog/manual/index.html>

