2. LIMBAJUL VERILOG HDL

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

Astfel, vor fi reliefate următoarele aspecte:

- Istoric, caracteristici generale şi funcţionale, structura şi fluxul de proiectare;
- Sintaxa şi semantica;
- Operatori, construcţii/instrucţiuni, task-uri şi funcţii;
- Sincronizare şi modelare, programe de test.

2.1. INTRODUCERE

2.1.1. ISTORIC

Verilog a fost dezvoltat în momentul când proiectanţii 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 porţi şi simulatoare funcţionale. Mai mult, majoritatea limbajelor tradiţionale de programare erau-sunt în esenţă secvenţiale şi astfel semantica lor era o provocare pentru modelarea concurenţei 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 substanţial până în 1987 şi redenumit **Verilog-XL**. **Verilog** a împrumutat mult de la limbajele existente, ca de exemplu: aspectele referitoare la concurenţă de la Modula şi Simula, sintaxa de la C şi metodele de combinare a nivelurilor de abstractizare de la HiLo (Brunnel University, UK). Limbajul nu era standardizat şi a suferit multe modificări până în 1990.

În anul 1989, *GDA* (precum şi drepturile asupra **Verilog** şi **Verilog-XL**) a fost cumpărată de către *Cadence* care a pus **Verilog** în domeniul public în 1990. Acest lucru a permis şi 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 specificaţiile pentru **VHDL** (Vhsic HDL) în anul 1983. Dezvoltat cu restricţii 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) şi a furnizat în anul 1991 documentaţia pentru **Verilog HDL.** În anul 1992, OVI (cunoscută acum ca *Accellera*) a dus o muncă laborioasă de îmbunătăţire a manualului de referinţă a limbajului (LRM – „Language Reference Manual") şi, 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 şi **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 îmunătăţiri ulterioare, printr-un proiect separat System Verilog, limbajul a devenit stahdard IEEE 1364-2005 sau **Verilog-2005.** Prin includerea suportului de modelare analogice şi 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 ŞI STRUCTURA UNUI PROGRAM VERILOG HDL

Verilog este un limbaj de descriere "hardware" (HDL) utilizat pentru a modela sisteme numerice. Limbajul suportă proiectare, verificare şi implementare a circuitelor analogice, digitale şi 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 şi limbajul C, **Verilog** are un pre-procesor, construcţii de control ca "if", "while", etc, rutine de afişare şi 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 biţi, nu are structuri, pointeri şi subrutine recursive (totuşi, **System Verilog** include acum aceste capabilităţi) şi lucrează cu conceptul de timp, important pentru sincronizare.

Limbajul diferă de un limbaj convenţional în sensul că execuţia instrucţiunilor nu este strict liniară. Un proiect **Verilog** constă într-o ierarhie de module. Modulele sunt definite ca un set de porturi de intrare, ieşire şi bidirecţionale. Intern, modulele conţin fire de legătură şi registre. Relaţia dintre porturi, fire şi registre este definită prin construcţii concurente şi secvenţiale care stabilesc comprtamentul modului. Construcţiile secvenţiale sunt plasate în interiorul blocurilor şi sunt executate în ordine secvenşială în cadrul blocului. Toate construcţiile concurente şi blocurile din proiect sunt executate în paralel. Un modul poate conţine şi una sau mai multe instanţe ale altor module definite în ierarhie.

Un subset de construcţii din limbaj sunt sintetizabile. Daca modulele din proiect conţin numai construcţii sintetizabile, programul **Verilog** poate fi utilizat pentru a sintetiza proiectul într-o listă de legături care descrie componentele de bază şi 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 biţi ("bitstream") pentru un dispozitiv logic programabil (cum este **FPGA –** "Field Programmable Gate Arrays").

Actualmente există o concurenţă puternică între limbajele **VHDL** şi **Verilog**, ceea ce impune prezentarea unei scurte comparaţii între cele două limbaje.

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

- **Verilog** este uşor de invăţat pentru ca **C** este mai simplu. El produce astfel mai mult cod compact, atât pentru citire, cât şi pentru scriere. Mai mult, proiectanţii (care deja ştiu **C** comparativ cu cei care ştiu **ADA**) îl învaţă şi fac "training" mai uşor.
- **VHDL** este foarte puternic tipizat şi permite programatorilor să-şi definească propriile lor tipuri deşi, în practică se utilizează tipurile de bază şi 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 faţă de **VHDL**:

- Permite modelarea la nivel de comutare.
- Se asigură faptul că toate semnalele sunt iniţializate ca nedefinite ceea ce asigură că toţi proiectanţii vor furniza logica necesară de iniţializare a proiectelor lor (tipurile de bază din **VHDL** iniţializează la 0).

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

- Se permit instanţieri condiţionale de module (*if/for ... generate*)
- Se furnizează un mecanism simplu (construcţia *configure*) care permite proiectantului să comute uşor între descrieri diferite de module particulare.

Alegerea între cele două limbaje nu se poate face însă în mod izolat. Trebuie incluşi şi alţi factori în mediul de proiectare, cum ar fi viteza de simulare, uşurinţa cu care se poate testa şi 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 proiectanţilor posibilitatea dezvoltării mediului de proiectare prin includerea de interfeţe grafice sau rutine de tip **C** pentru a calcula întârzierile în analiza sincronizării. Pragmatic, un proiectant ar fi bine să le ştie 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 tradiţională. Fiecare proiect este realizat la nivel de porţi utilizând porţi 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 intâi 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.

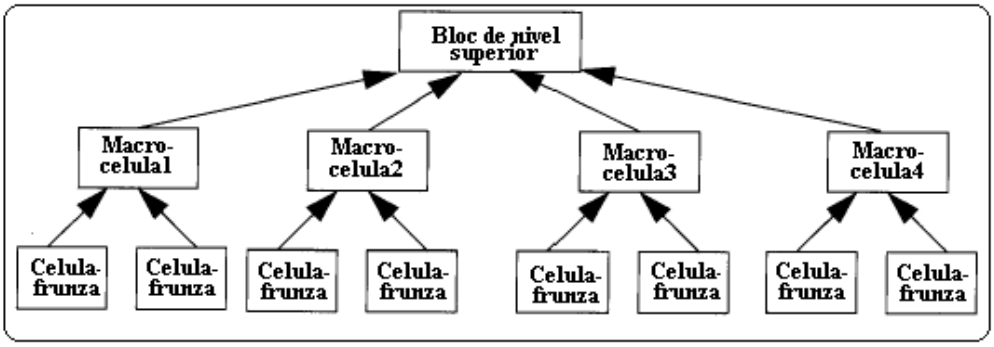


Figura 2.1. Ierarhia de blocuri "bottom-up"

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

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

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

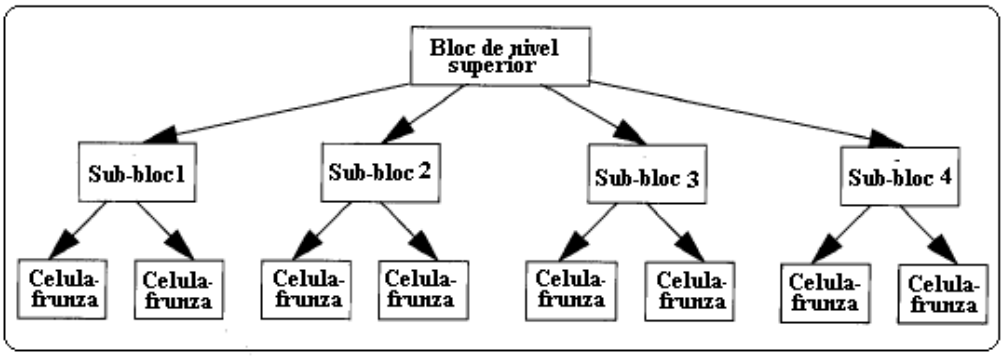


Figura 2.2. Ierarhia de blocuri “top-down”

Structura unui program Verilog

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

În general, există un modul pe nivelul cel mai înalt, care specifică un sistem închis ce conţine, atât datele de test, cât şi modelele/modulele hardware. Modulul de pe nivelul cel mai înalt invocă instanţe ale celorlalte module; acest modul poate fi împărţit într+un modul de test şi unul sau mai multe module principale, care conţin celelalte submodule.

Modulele reprezintă părţi “hardware”, care pot fi de la simple porţi până la sisteme complete, ca de exemplu un microprocesor. Ele pot fi specificate, fie comportamental, fie structural (sau o combinaţie a celor două). O **specificare comportamentală** defineşte comportarea unui sistem numeric (modul) folosind construcţiile limbajelor de programare tradiţionale, de exemplu: **if** şi **instrucţiuni 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 porţi, cât şi 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 afişează 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 ŞI SITEMELOR NUMERICE – NIVELURI DE ABSTRACTIZARE VERILOG

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

Modalităţi de descriere:	Comportamentală	Structurală	Fizică	Abstractă
Niveluri ierarhice:	Aplicaţii, sistemul de operare	Calculator, procesor	Modul, plachetă, circuit	Arhitectură, algoritm, modul/bloc funcţional
	Programe	Sumatoare, porţi, registre	Celule	Logică, comutaţie
	Subrutine, instrucţiuni	Tranzistoare	Tranzistoare	Circuit

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

1. Arhitectura, nivelul cel mai de sus, se referă la aspectele funcţionale ale sistemului. La acest nivel există următoarele trei subniveluri:

    - arhitectura sistemului: subnivelul cel mai înalt, care constituie interfaţa cu utilzatorul;
    - organizarea sistemului: structurarea sistemului în unităţi funcţionale, interconectarea lor, fluxul de informaţii dintre ele;
    - microarhitectura: structura unităţilor funcţionale, care realizează interfaţa dintre "hardware" şi "firmware".

2. Algoritmul mapează nivelul arhitectural la nivelul logic.
3. Modulul/blocul funcţional este un nivel tipic de transfer între registre (RTL – "Register Transfer Level"), care caracterizează în termeni ca registre, întârzieri, numărătoare, ceas, memorie, circuite combinaţionale şi secvenţiale.
4. Logica reprezintă nivelul care descrie sistemul, utilizând porţi logice.
5. Comutaţia 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.

Iniţial 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 şi 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 obţine unele capabile să înţeleagă cât mai multe niveluri de abstractizare, mergând până la nivelul logic.

Primele dintre ele au fost MIMOLA şi M, dar cele care reprezintă la ora actuală vârful în domeniul proiectării automate a structurilor "hardware" sunt VHDL ("Very High Speed Integrated Cicuit Hardware Description Language") şi 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 şi nivelul fizic.

- **Nivelul comportamental**

Nivelul comportamental descrie un sistem prin algoritmi concurenţi. Fiecare algoritm este el însuşi secvenţial, ceea ce înseamnă că el constă dintr-un set de instrucţiuni care se execută una după cealaltă. Elementele principale sunt funcţiile, “task”-urile şi blocurile “initial” şi “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 operaţii şi transferuri de date între registre. Se utilizează un semnal explicit de ceas. Proiectele RTL conţin delimitări exacte de întârzieri: operaţiile sunt planificate să apară la anumite momente de timp. O definiţie 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 şi întârzierile lor. Toate semnalele pot avea doar valori logice de tipul “0”, “1”, “X” şi “Z”. Operaţiile uzuale sunt predefinite ca primitive logice (porţi 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 porţi.

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)**

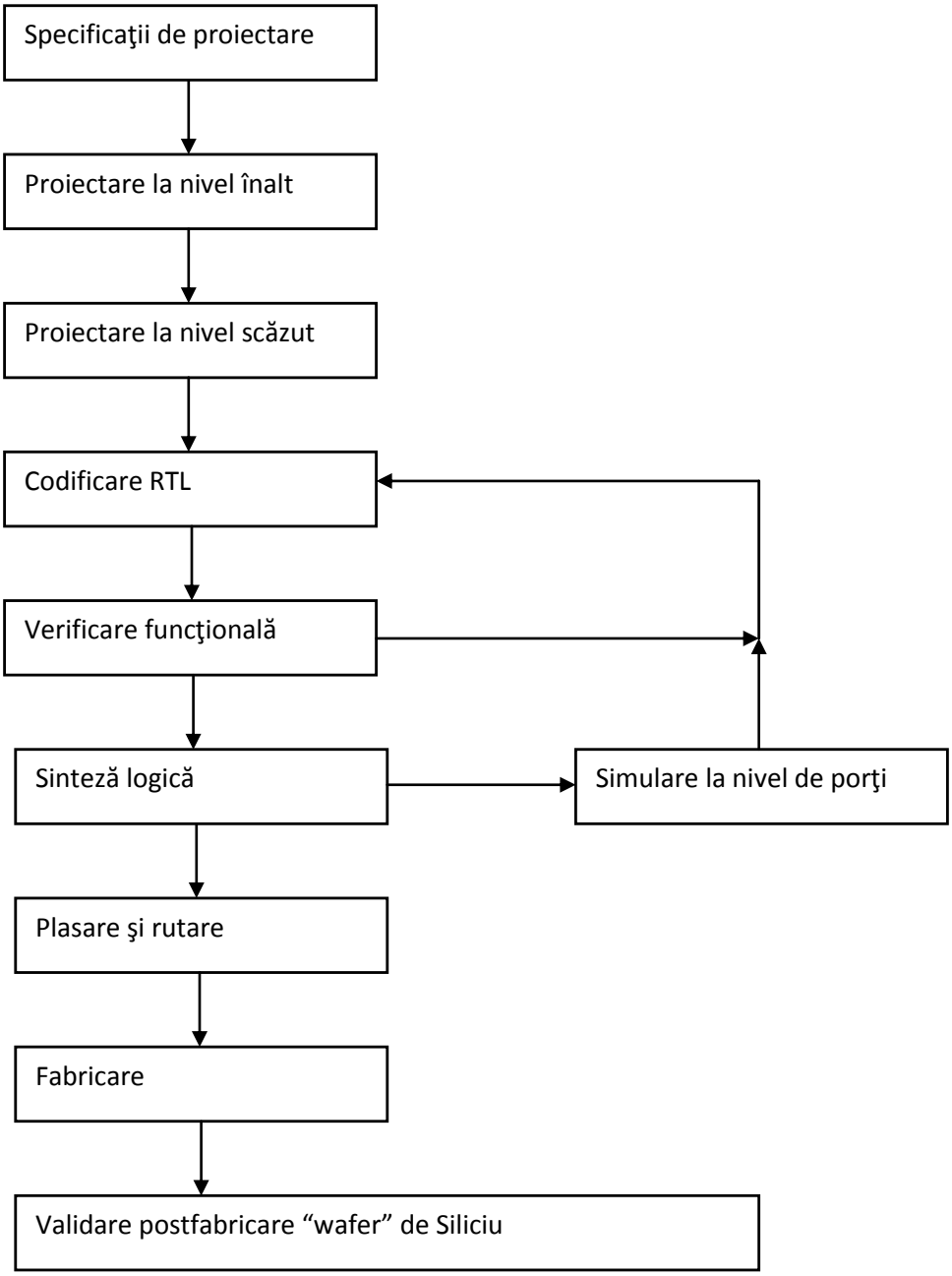
Descrierea fizică furnizează informaţii privind modul de construcţie al unui circuit particular descris din punct de vedere comportamental şi structural; descrierea fizică este asociată cu descrierea la nivel de măşti tehnologice. Această descriere cuprinde mai multe niveluri de abstractizare: nivelul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

componentei pentru care se specifică poziţia, stratul, numele şi lăţimea) iar la nivelul cel mai de jos se află apelările de tranzistoare, fire şi la conexiuni.

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

2.1.4. FLUXUL DE PROIECTARE

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



Specificaţii de proiectare

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

Proiectarea la nivel înalt

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

Proiectarea la nivel scăzut

Proiectarea aceasta se mai numeşte şi microproiectare şi este faza în care proiectantul descrie implementarea fiecărui bloc. Ea conţine detalii despre maşina de tranziţii de stări şi despre blocurile funcţionale. Este bine să se deseneze formele de undă pentru diferitele interfaţări. La această fază se pierde destul de mult timp. Se pot utiliza editoare de texte şi 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"-utilor folosite la proiectare.

Verificarea şi simularea

Această etapă este procesul de verificare funcţională la orice nivel de abstractizare. Se utilizează simulatoare adecvate. Pentru a testa dacă codul RTL îndeplineşte cerinţele funcţionale din specificaţiile de proiectare, trebuie să se scrie programe de test ("testbench") cu care se va verifica funcţional codul RTL proiectat pe baza studiului formelor de undă obţinute în urma simulării. Se pot utiliza simulatoare/compilatoare 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 porţi, cât şi la nivel de fire), după faza de sinteză, se mai realizează o simulare la nivel de porţi sau SDF ("Standard Delay Format").

Sinteza

Este procesul prin care "tool"-urile de sinteză ca XST, Ahdl sau Synplify iau codul RTL şi, ţinând cont de tehnologie şi constrângeri ca intrări, îl mapează în primitive tehnologice la nivel de porţi. "Tool"-ul de

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

Plasarea şi rutarea

Lista de legături la nivel de porţi generată de "tool"-ul de sinteză este luată şi importată în "tool"-ul de plasare şi rutare în format listă de legături **Verilog.** Mai întâi toate porţile şi bistabilii sunt plasaţi iar semnalele de ceas şi iniţializare sunt rutate, după care fiecare bloc este rutat obţinându-se ca ieşire un fişier GDS utilizat la fabricarea circuitelor ASIC.

Validare postfabricare

O dată ce circuitul este gata de fabricaţie, este necesar ca el să fie pus în condiţii de mediu real de funcţionare şi testat înainte de a fi lansat pe piaţă. Aceasta deoarece viteza de simulare a RTL este scăzută şi există posibilitatea găsirii unor "bug"-uri la testarea "wafer"-elor de Siliciu.

2.2. SINTAXA ŞI SEMANTICA LIMBAJULUI VERILOG HDL

2.2.1. CONVENŢII LEXICALE

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

Spaţii libere (albe)

Spaţiile albe pot conţine caractere precum spaţii (/b – "blank"), caractere "tab" (/t), caractere linie nouă (/n), caractere sfârşit de linie ("Carriage Return" - <CR>), sfârşit de paragraf şi sfârşit de fişier (EOF – "End Of File").

Aceste caractere sunt ignorate cu excepţia când servesc la separarea altor convenţii sau în cadrul şirurilor.

Comentarii

Există două moduri de a introduce comentarii:

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

Identificatori

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

Identificatorii trebuie să înceapă cu un caracter alfabetic sau caracterul "underscore" (_). Ei pot conţine caractere alfabetice, caractere numerice, _ şi $. 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 şi terminând numele lui cu un spaţiu 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 impedanţă (valoare trei-stări sau flotantă)
- X sau x - nedefint, necunoscut sau neiniţializat

Puteri logice

În **Verilog** există următoerele tipuri de puteri logice (valori logice puternic cuplate – "logic strengths"):

Nivel de putere	Nume valoare puternic cuplată	Mnemonică specifică		Afişare mnemonică	
7	"Drive" de alimentare	supply0	supply1	Su0	Su1
6	"Drive" puternic cuplat	strong0	strong1	St0	St1
5	"Drive" ţinut („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	Impedanţă 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 şi 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 şi valoare sunt permise spaţii
- <baza> (opţional) este baza de numeraţie (binară, octală, zecimală sau hexazecimală). Ea este predefinită ca zecimală şi nu este "case-sensitive"
- <mărime> (opţional) este numărul de biţi din întreg. Întregii fără semn sunt predefiniţi la 32 biţi
- <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 şi este ignorat
- Valorile sunt expandate de la dreapta la stânga (lsb ("least signifiquant bit") spre msb ("most signifiquant bit"))
- Când mărimea este mai mică decât valoarea, atunci biţii cei mai semnificativi sunt trunchiaţi. Când mărimea este mai mare decât valoarea, atunci biţii cei mai semnificativi sunt setaţi la 0 dacă valoarea este 0 sau 1, la Z sau X dacă valoarea este Z sau X
- Valorile '0' şi '1' pot fi folosite ca numere pentru că nu pot fi identificatori
- Numerele pot fi declarate ca parametri

Numere reale

Sintaxa este:

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

<valoare mantisă>E<valoare exponent> unde E nu este “case-sensitive” (notaţia ştiinţifică)

Caracteristici:

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

2.2.3. MODULE; INSTANŢIERI, PARAMETRIZĂRI ŞI IDENTIFICATORI IERARHICI

Definirea modulelor în Verilog HDL

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

Sintaxa unui modul este următoarea:

Sintaxa
Conexiune internă implicit (prin ordinea porturilor) **module** nume_modul (nume_port, nume_port, ...); componente_modul **endmodule**
Conexiune internă explicit (prin nume) **module** nume_modul (.nume_port (nume_semnal), .nume_port (nume_semnal), ...); componente_modul **endmodule**

- Conexiunea internă implicită conectează portul la un fir intern sau un registru cu acelaşi 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 şters.
- Conexiunea internă explicită conectează portul la un semnal intern cu nume diferit sau la semnale interne selectate pe biţi sau pe părţi sau concatenate.
  Numele trebuie să se potrivească cu modulul frunză iar ordinea nu este importantă.
- Componentele unui modul sunt:
  - declaraţii_porturi_modul
  - declaraţii_tipuri de date
  - instanţieri_module
  - instanţieri_primitive
  - blocuri_procedurale
  - atribuiri_continue
  - definiţii_task-uri
  - definiţii_funcţii
  - blocuri_specificate
- Funcţionalitatea unui modul poate fi:
  - Comportamentală (RTL): modelat cu constructori de blocuri procedurale sau atribuiri continue
  - Structurală: modelat ca o listă de legături de instanţe de module şi/sau primitive
  - Combinată: comportamental şi structural
- Definiţiile de module nu pot fi îmbricate însă un modul poate instanţia alte module.
- Definiţiile de module pot fi exprimate utilizând constante parametru.
  Fiecare instanţă de modul poate redefini parametrii pentru a fi unici pentru acea instanţă.
- Componentele unui modul pot apărea în orice ordine dar declaraţiile de porturi şi cele de tipuri de date trebuie să apară înainte ca porturile sau semnalele să fie referite.

Porturi

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

Sintaxa de declarare a porturilor este următoarea:

Sintaxa
direcţie_port [msb : lsb] nume_port, nume_port, ... ;

- Direcţia porturilor poate fi:
  - **input** pentru scalari sau vectori de intrare
  - **output** pentru scalari sau vectori de ieşire
  - **inout** pentru scalari sau vectori bidiracţionali
- Valorile msb şi lsb trebuie să fie întregi, parametri întregi sau o expresie care se rezumă la o constantă întreagă.

  Se poate folosi atât convenţia “little-endian” (lsb este cel mai mic bit) cât şi convenţia “big-endian” (lsb este cel mai mare bit)
- Dimensiunea maximă a unui port poate fi limitată dar nu mai puţin de 256 biţi
- 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
  - Ieşirile: intern pot fi de tip fir sau registru, extern trebuie să fie conectate la o variabilă de tip registru
  - Porturile bidirecţionale: intern şi extern trebuie întotdeauna să fie de tip fir şi pot fi conectate numai la o variabilă de tip fir
  - Potrivirea ca dimensiuni: este legal să se conecteze intern şi extern porturi de dimensiuni diferite; dar atenţie că tool-urile de sinteză pot raporta probleme
  - Porturi neconectate: sunt permise prin utilizarea notaţiei “,”
  - 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

Instanţierea modulelor

Sintaxa pentru instanţiere este următoarea:

Sintaxa
Conexiuni implicite prin ordinea porturilor nume_modul nume_instanţă [domeniu_tablou_instanţe] (semnal, semnal, ...);
Conexiuni explicite prin numele porturilor nume_modul nume_instanţă [domeniu_tablou_instanţe] (.nume_port(semnal), (.nume_port(semnal), ...);
Redefinire explicit prin parametri **defparam** cale_ierarhică.nume_parametru = valoare;
Redefinire implicit prin parametri nume_modul **#**(valoare sau nume parametru) nume_instanţă (semnale);

- Un modul poate fi instanţiat utilizând ordinea sau numele porturilor:
  - Instanţierea prin ordinea porturilor listează semnalele conectate la ele în aceeaşi ordine ca în lista de porturi din definirea modulului
  - Instanţierea prin numele porturilor listează numele portului şi semnalul conectat la el în orice ordine
- *Nume_instanţă* (cerut) este utilizat pentru e realiza instanţe multiple ale aceluiaşi modul în mod unic dintr-un altul.
- *Domeniu_tablou_instanţe* (opţional) instanţiază module multiple, fiecare instanţă fiind conectată la biţi separaţi ai unui vector:
  - Domeniul este specificat ca [lhi:rhi] (index stânga (“left-hand-index”) la index dreapta (“right-hand-index”))
  - Dacă dimensiunea pe biţi a unui modul port dintr-un tablou este aceeaşi cu cea a semnalului conectat la el, semnalul complet este conectat la fiecare instanţă a modulului port; daca dimensiunea diferă, atunci fiecare instanţă de modul port este conectată la o parte selectată a semnalului, cu cel mai puţin semnificativ index de instanţă conectat la cea mai puţin semnificativă parte a unui vector şi tot aşa progresiv către stânga
  - Trebuie să existe un număr corect de biţă în fiecare semnal pentru a se conecta la toate instanţele (mărimea semnalului şi mărimea portului trebuie să fie multiple)

- Parametrii dintr-un modul pot fi redefiniţi pentru fiecare instanţă:
    - Redefinirea explicită utilizează o construcţie **defparam** cu numele ierarhic al parametrului
    - Redefinirea implicită utilizează jetonul **#** ca parte a instanţierii modulului. Parametrii trebuie redefiniţi în aceeaşi ordine în care au fost declaraţi prin modul
- Numele căii ierarhice este dat de identificatorul modulului rădăcină (“top-level”) urmat de idntificatorii instanţelor modulului, separate prin puncte. Acest lucru este folositor atunci când se doreşte a se vedea semnalul dintr-un modul frunză sau când se doreşte forţarea unei valori înăuntrul unui modul intern.

2.2.4. PRIMITIVE PROGRAM

Primitive

Verilog are primitive incluse (”built-in”) cum sunt porţile, porţile de transmisie şi 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 porţi, sau cum se mai numeşte, simularea SDF (“Standard Delay Format”). De asemenea, formatul de ieşire al listei de legături din tool-ul de sinteză, care este importat în tool-ul de plasare şi de rutare, este tot lcu primitive **Verilog** la nivel de porţi. Sintaxa de utilizare a acestor primitive este următoarea:

Poartă	Descriere	List de terminale
and **or** **xor**	Poartă ŞI Poartă SAU Poartă SAU-EXCLUSIV	(1 ieşire, 1 sau mai multe intrări)
nand **nor** **xnor**	Poartă ŞI-NU Poartă SAU-NU Poartă SAU-NU-EXCLUSIV	(1 ieşire, 1 sau mai multe intrări)

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

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

Instanţierea primitivelor

Sintaxa de instanţiere este următoarea:

Sintaxă primitivă
tip_poartă (*drive_strength*) **#**(*întârziere*) nume_instanţă [domeniu_instanţă_tablou] (*terminal*, *terminal*, ...);
tip_comutator #(*întârziere*) nume_instanţă [domeniu_instanţă_tablou] (*terminal*, *terminal*, ...);

iar sintaxa întârzierilor primitivelor este:

Sintaxă întârziere primitivă
#întârziere or **#**(întârziere) *Întârziere singulară pentru toate tranziţiile de ieşire*
#(întârziere, întârziere) *Întârzieri separate pentru tranziţii (rising, falling)*
#(întârziere, întârziere, întârziere) *Întârzieri separate pentru tranziţii (rising, falling, turn-off)*
#(întârziere_minimă:întârziere_tipică:întârziere_maximă) *Domeniu de întârzieri de la minimum la maximum pentru toate tranziţiile*

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

unde:

- Întârzierea “rise” este asociată tranziţiei ieşirii porţii în 1 din 0, x, z
- Întârzierea “fall” este asociată tranziţiei ieşirii porţii în 0 din 1, x, z
- Întârzierea “turn-off” este asociată tranziţiei ieşirii porţii în z din 0, 1, x
- Întârzierea (opţional) reprezintă timpul de propagare prin primitivă şi este predefinită la zero; poate avea valori întregi sau reale
- Nume_instanţă (opţional) poate fi utilizat pentru a referi primitive specifice în tool-urile de depanare, scheme, etc
- Domeniu_instanţă_tablou (opţional) instanţiază primitive multiple, fiecare instanţă fiind conectată la biţi separaţi ai unui vector, unde semnalele vector trebuie să aibă aceeaşi dimensiune cu cea a tabloului iar semnalele scalare sunt conectate la toate instanţele din tablou; domeniul este specificat ca [lhi:rhi] la fel ca la instanţierea modulelor

Primitive Definite de Utilizator

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

UDP pot modela logică combinaţională şi secvenţială şi pot include întârzieri. Ele trebuie definite în afara corpului **module** ... **endmodule**.

Sintaxan UDP este următoarea:
primitive nume_primitivă (porturi/terminale);
declaraţii_primitive …
construcţii_**initial** ...
tabelă_definiţie
endprimitive

unde:

- nume_primitivă: identificator
- declaraţii_UDP: declaraţii_ieşiri_UDP, declaraţii_registre pentru ieşiri (numai pentru UDP secvenţiale), declaraţii_intrări_UDP;
- construcţii **initial** (opţional): stabilesc valoarea iniţială (1'b0, 1'b1, 1'bx, 1, 0) a terminalelor de ieşire prin atrinuirea unei valori literale pe un singur bit unui registru terminal de ieşire; valoarea predefinită este "x".
- tabela_definiţie a funcţionării: descrie funcţionalitatea primitive, atât combinaţional, cât şi secvenţial şi are structura **table** … intrări_tabelă ... **endtable**;

Caracteristicile UDP:

- Reguli de folosire a porturilor în UDP:
    - O UDP poate conţine numai o ieşire şi până la 9 intrări (UDP secvenţiale) sau 10 intrări (UDP combinaţionale); portul de ieşire 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 bidirecţionale
    - Terminalul de ieşire a unei UDP secvenţiale cere o declaraţie suplimentară ca tip registru
    - Este ilegal să se declare un registru pentru un terminal de ieşire a unei UDP combinaţionale
- Fiecare linie dintr-tabelă este o condiţie; când o intrare se schimbă, condiţia de intrare este marcată şi ieşirea este evaluată pentru a reflecta noua schimbare a intrării
- UDP utilizează simboluri speciale pentru a descrie funcţii ca front crecător, nu contează ş.a:

Simbol	Interpretare	Explicaţie
0 sau 1 sau x sau X	0 sau 1 sau x sau X	0, 1 sau necunoscut pe intrare sau ieşire
?	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 şi z
n sau N	(10) sau (1x) sau (x0) sau (0z) sau (z0)	Front negativ incluzând x şi z
*	(??)	Toate tranziţiile posibile pe intrări
-	Nici o schimbare	Nici o schimbare la ieşire (numai pentru UDP secvenţiale)

- **UDP combinaţionale**

Ieşirea este determinată ca o funcţie de intrarea curentă. Ori de câte ori o intrare îşi schimbă valoarea, UDP este evaluată şi unul dintre rândurile tabelei de stare este marcat. Ieşirea stării este setată la valoarea indicată prin acel rând. Acest lucru este similar construcţiilor de condiţie: fiecare linie din tabelă este o condiţie. UDP combinaţionale au un câmp pentru intrări şi unul pentru ieşire, care sunt separate prin simbolul“**:**”. Fiecare rând al tabelei este terminat prin simbolul “**;**”. Ordinea intrărilor în descrierea tabelei de stare trebuie să corespundă cu ordinea intrărilor din lista de porturi din antetul de definire a UDP. Nu are legătură cu ordinea de declarare a intrărilor. Intrările trebuie separate unele de altele printr-un spaţiu liber.

Fiecare rând din tabelă defineşte ieşirea pentru combinaţie particulară de stări de intrare. Dacă toate intrările sunt specificate ca “x”, atunci ieşirea trebuie specificată ca **”x”**. Toate combinaţiile care nu sunt specificate explicit rezultă în starea predefinită de ieşire ca ”x”.

Este ilegal sa existe aceeaşi combinaţie de intrări, specificată pentru diferite ieşiri.

- **UDP secvenţiale senzitive pe nivel**

Comportarea este specificată la fel ca mai sus, cu excepţia faptului că ieşirea este declarată ca tip registru şi există un câmp suplimentar în fiecare intrare a tabelei, care reprezintă starea curentă a UDP.

Ieşirea indică faptul că există o stare internă şI are aceeaşI valoare cu ea. Câmpul suplimentar este separat prin simbolul “**:**” între câmpul pentru intrări şi cel pentru ieşire.

- **UDP secvenţiale senzitive pe front**

Diferă faţă de cele anterioare prin faptul că schimbările la ieşire sunt eşantionate prin tranziţii specifice ale intrărilor. Pentru fiecare intrare în tabelă, numai un singur semnal de poate avea poate avea specificată o tranziţie pe front. Toate celelalte semnale de intrare trebuie să aibă intrări în tabelă pentru a acoperi tranziţiile sau UDP va furniza “x” la ieşire când tranziţia apare. Intrările din tabelă senzitive pe front sunt mai puţin prioritare decât cele senzitive pe nivel.

2.2.5. TIPURI DE DATE; ŞIRURI

Sintaxa de declarare a tipurilor de date este următoarea:

Sintaxa
tip_registru [dimensiune] *nume_variabilă* , *nume_variabilă* , ... ;
tip_registru [dimensiune] *nume_memorie* [dimensiune_tablou];
tip_legătură [dimensiune] #(întârziere) *nume_legătură* , *nume_legătură* , ... ;
tip_legătură (“drive_strength”) [dimensiune] #(întârziere) *nume_legătură* = atribuire_continuă;
trireg (”capacitance_strength”) [dimensiune] #(întârziere, timp_decay) *nume_legătură*, *nume_legătură*, ... ;
parameter *nume_constantă* = *valoare*, *nume_constantă* = *valoare*, ... ;
specparam *nume_constantă* = *valoare*, *nume_constantă* = *valoare*, ... ;
event *nume_eveniment*, *nume_eveniment*, ... ;

unde:

- Întârziere (opţional) poate fi specificată numai pentru tipul legătură. Sintaxa este la fel ca la primitive.
- Dimensiunea este un domeniu de forma [msb : lsb].

  Valorile msb şi lsb trebuie să fie întregi, parametri întregi sau o expresie care se rezumă la o constantă întreagă. Se poate folosi atât convenţia “little-endian” (lsb este cel mai mic bit) cât şi convenţia “big-endian” (lsb este cel mai mare bit).

  Dimensiunea maximă a unui vector poate fi limitată dar nu mai puţin de 2^{16} biţi.
- 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 puţin de 2^{24} biţi.
- Timp_decay (opţional) specifică volumul de timp în care o legătură **trireg** va memora o încărcare după ce toate sursele se vor opri, înainte de ajungerea la vcaloarea 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 declaraţie în codul **Verilog**
- Prin declarare implicită: fără declarare când se conectează structural blocurile constructive din cod; aceasta este întotdeauna o legătură de tip “wire” şi de dimensiune un bit.

Tipuri legătură

Fieacre tip de legătură are o funcţionalitate care este utilizată pentru a modela diferite tipuri de “hardware” (cum ar fi PMOS, NMOS, CMOS, etc).

Există următoarele tipuri de legături:

Mnemonica	Funcţionalitatea
wire sau **tri**	Interconectare simplă de fire
wor sau **trior**	Ieşiri legate prin SAU cablat (model ECL)
wand sau **triand**	Ieşiri legate prin ŞI 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”)
trireg	Reţine 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 trireg 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 şi puteri (“strengths”) logice.

O legătură de date trebuie utilizată când:

- un semnal este dat de către ieşirea unui anumit dispozitiv
- un semnal este declarat ca port de intrare sau bidirecţional
- 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 reţele 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	Funcţionalitatea
reg	Variabilă fără semn pe orice dimensiune de biţi
integer	Variabilă cu semn pe 32 biţi
time	Variabilă fără semn pe 64 biţi
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”) şi trebuie utilizate când semnalul este pe stânga unei atribuiri procedurale.

Şiruri

Un şir este o secvenţă de caractere încadrate prin ghilimele şi toate conţinute într-o singură linie. Şirurile utilizate ca operanzi în expresii şi atribuiri sunt tratate ca o secvenţă de valoare ASCII pe 8 biţi, valoare ce reprezintă un caracter. Pentru a declara o variabilă să memoreze un şir, se declară un registru suficient de mare pentru a ţine numărul maxim de carcatere pe care variabila îl păstrează; în **Verilog** nu se cer biţi suplimentari pentru a păstra o terminaţie de caracter. Şirurile pot fi manipulate utilizând operatori standard. Când o variabilă este mai mare decât este cerută să păstreze o valoare în timpul atribuirii, **Verilog** umple conţinutul la stânga cu zerouri după atribuire. Acest fapt asigură consistenţa în timpul atribuirii unor valori non-şir.

Anumite caractere pot fi utilizate în şiruri 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 digiţi 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
```

<u>Alte tipuri de date</u>

Alte tipuri	Funcţionalitate
parameter	Constantă “run-time” pentru memorarea întregilor, numerelor reale, timpului, întârzierilor sau şirurilor ASCII. Parametrii pot fi redefiniţi pentru fiecare instanţă a modulului.
specparam	Specifică constanta bloc pentru memorarea întregilor, numerelor reale, timpului, întârzierilor sau şirurilor ASCII.
event	Un “flag” temporar fără valoare logică sau memorare date. Este utilizat adesea pentru sincronizarea activităţilor concurente dintr-un modul.

2.3. OPERATORI DE LIMBAJ

Operatorii execută o operaţie pe unul sau doi operanzi:

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

Operanzii pot fi de tip legături sau registru şi pot fi scalar, vector sau o selecţie de biţi 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 şi 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 faţă de 2)
*	m * n	Înmulţeşte m cu n
/	m / n	Împarte m la n
%	m % n	Modulul de m / n

Împărţirea întreagă trunchiază orice parte fracţională. Rezultatul operaţiei *modul* ia semnul primului operand. Daca 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
~	~m	Inversează (neagă) fiecare bit al lui m
&	m & n	ŞI 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 ^ n	SAU EXCLUSIV fiecare bit al lui m cu fiecare bit al lui n
~^ ^~	m ~^ n m ^~ 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 potriveşte ca lungime cu operandul mai lung. Operaţiile consideră şi biţi de valoare “x”.

2.3.3. OPERATORI DE REDUCERE

Simbol	Utilizare	Descriere
&	&m	ŞI logic împreună pe toţi biţii din m (1 - Adevărat/1 - Fals)
~&	~&m	ŞI-NU logic împreună pe toţi biţii din m (1 - Adevărat/1 - Fals)
|	|m	SAU logic împreună pe toţi biţii din m (1 - Adevărat/1 - Fals)
~|	~|m	SAU-NU logic împreună pe toţi biţii din m (1 - Adevărat/1 - Fals)
^	^m	SAU EXCLUSIV împreună pe toţi biţii din m (1 - Adevărat/1 - Fals)
~^ ^~	~^m ^~m	SAU-NU EXCLUSIV împreună pe toţi biţii din m (1 - Adevărat/1 - Fals)

Operatorii de reducere sunt unari şi produc rezultat pe 1 bit. Operaţiile de reducere consideră şi biţii de valoare ”x”.

2.3.4. OPERATORI LOGICI

Simbol	Utilizare	Descriere
!	!m	m negat logic (1 - Adevărat/1 - Fals)
&&	m && n	m ŞI 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 conţine biţi 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 conţin “x” sau “z”) iar identitatea este cazuistică (include şi comparaţii de “x” şi “z”).

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

2.3.6. OPERATORI RELAŢIONALI

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 <= n	Este m mai mic sau egal decât n? (1 - Adevărat/1 - Fals)
>=	m >= 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. ALŢI OPERATORI

Operatori deplasare logică		
Simbol	**Utilizare**	**Descriere**
<<	m << n	Deplasare m la stânga de n-ori
>>	m >> n	Deplasare m la dreapta de n-ori

Poziţiile vacante sunt umplute cu zerouri.

Miscellaneous Operators		
Simbol	**Utilizare**	**Descriere**
? :	sel?m:n	Operator condiţional: 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	Poziţionează 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 operaţii cu operatori de concatenare şI replicare îmbricaţi (exemplu: -> {a, {4{b, c}}} // echivalent cu {a, b, c, b, c, b, c, b, c}).

2.3.8. PRECEDENŢA OPERATORILOR

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

Operatorii de pe aceeaşi linie au aceeaşi precedenţă şi asociază termenul din stânga celui din dreapta într-o expresie.

Parantezele sunt utilizate pentru a modifica precedenţa sau pentru a clarifica situaţia.

Pentru a uşura înţelegerea, se recomandaă utilizarea parantezelor.

2.4. CONSTRUCŢII/INSTRUCŢIUNI VERILOG

2.4.1. BLOCURI PROCEDURALE

De regulă codul comportamental **Verilog** există în interiorul blocurilor procedurale, dar există şi excepţii 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 porneşte execuţia)
- **always**: aceste bucle se execută mereu şi mereu, adică, după cum sugerează şi numele, se execută întotdeauna pâna la oprirea simulării

Sintaxa blocurilor procedurale este următoarea:

Sintaxa
tip_bloc @(listă_senzitivitate) grup_instrucţiuni: nume_grup declaraţii_variabile_locale control_sincronizare instrucţiuni_procedurale sfârşit_grup_instrucţiuni

unde:

- tip_bloc: este unul din cele două tipuri de blocuri procedurale
- listă_senzitivitate: este un eveniment de control al sincronizării care stabileşte când vor începe a fi evaluate toate instrucţiunile din blocul procedural; lista este utilizată pentru a modela comportarea logică combinaţională şi secvenţială
- structura grup_instrucţiuni - sfârşit_grup_instrucţiuni: este utilzată pentru a grupa împreună două sau mai multe instrucţiuni procedurale şi controlează ordinea de execuţie
    - **begin – end** grupează împreună două sau mai multe instrucţiuni în mod secvenţial, astfel încât instrucţiunile să fie evaluate în ordinea în care sunt listate; fiecare control de sincronizare este relativ la instrucţiunea precedentă iar blocul se termină după ultima instrucţiune din bloc
    - **fork – join** grupează împreună două sau mai multe instrucţiuni în mod paralel, astfel încât toate instrucţiunile sunt evaluate concurent; fiecare control de sincronizare este absolut la momentul când grupul începe să se execute iar blocul se termină când după ultima instrucţiune (instrucţiunile cu cea mai mare întârziere pot fi primele din bloc)

- Pot exista construcţii **fork – join** în interiorul unei construcţii **begin – end**
- nume_grup (opţional): crează un scop în grupul de instrucţiuni; grupurile (blocurile) denumite pot avea variabile locale şi pot fi dezactivate prin construcţia **disable**
- declaraţii_variabile_locale (opţional): trebuie să fie un tip de date registru
- control_sincronizare: este utilzat pentru a controla atunci când se execută instrucţiunile dintr-un bloc procedural; va fi prezentat ulterior
- instrucţiuni_procedurale: sunt atribuiri procedurale la o variabilă registru (**reg, integer,** time) sau instrucţiuni program şi vor fi prezentate ulterior; atribuirile procedurale pot atribui valori de tip legătură (**wire**), constante, un alt registru sau o valoare specific iar instrucţiunile program pot fi instrucţiuni condiţionale, de caz, de repetare sau dezactivare

2.4.2. ATRIBUIRI PROCEDURALE

Pot fi atribuiri de tip blocante sau nonblocante.

Atribuirile blocante se execută în ordinea în care au fost scrise, adică sunt secvenţiale.Ele au simbolul "=". Aceste atribuiri acţionează ca şi în limbajele tradiţionale de programare: instrucţiunea următoare nu se execută pâna ce întreaga instrucţiune curentă este complet executată şi doar atunci se trece controlul execuţiei la instrucţiunea următoare celei curente.

Sintaxa unei atribuiri blocante este următoarea:
nume_tip_date_registru = *expresie*;

Atribuirile nonblocante sunt executate în paralel. Ele au simbolul "<=". Instrucţiunea următoare nu este blocată în timpul execuţiei instrucţiunii curente. Atribuirea nonblocantă evaluează termenul din dreapta, pentru unitatea curentă de timp şi atribuie valoarea obţinută termenului din stânga, la sfârşitul unităţii de timp.

Sintaxa unei atribuiri nonblocante este următoarea:
nume_tip_date_registru <= *expresie*;

Exemplu:

```
//se presupune că iniţial a = 1
// testarea atribuirii blocante
```

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

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

      end               //rezultat final a=3
```

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

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

Controlul sincronizării va fi prezentat ulterior.

Există următoarele tipuri de atribuiri procedurale, blocante sau nonblocante:

Assign şi deassign

Aceste construcţii de atribuire procedurală permit ca atribuirile continue să fie plasate pe registre pentru controlul perioadelor de timp. Construcţia **assign** se extinde peste atribuirile procedurale la o variabilă registru. Construcţia **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 şi release

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

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

De asemenea, pot fi utilizate pentru a insera unul sau doi biţi 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. CONSTRUCŢII/INSTRUCŢIUNI CONDIŢIONALE

Construcţia **if – else** controlează execuţia altor instrucţiuni şi poate include sincronizări procedurale.

Când mai mult de o instrucţiune necesită a fi executată pentru o condiţie **if**, atunci se vor utiliza **begin** şi **end** în interiorul construcţiei.

Sintaxa instrucţiunilor condiţionale este următoarea:

Sintaxa
Instrucţiune if **if** (*expresie_condiţie*) // dacă expresia evaluată este adevărată   *instrucţiune sau grup de instrucţiuni* // execută instrucţiunea sau grupul de instrucţiuni

Instrucţiune if – else
 	
if (*expresie_condiţie*) // dacă expresia evaluată este adevărată
  instrucţiune sau grup de instrucţiuni // execută instrucţiunea sau grupul de instrucţiuni
else // altfel dacă expresia evaluată este falsă sau necunoscută
  instrucţiune sau grup de instrucţiuni // execută instrucţiunea sau grupul de instrucţiuni

Instrucţiune îmbricată if – else – if
 	
if (*expresie1_condiţie*) // dacă expresia evaluată este adevărată
  instrucţiune sau grup de instrucţiuni // execută instrucţiunea sau grupul de instrucţiuni
else if (*expresie2_condiţie*) // altfel dacă expresia1 evaluată este falsă sau necunoscută
  //şi dacă expresia2 evaluată este adevărată
  instrucţiune sau grup de instrucţiuni // execută instrucţiunea sau grupul de instrucţiuni

În mod normal nu se include verificarea de ***reset*** în *expresia_condiţie.* Aşa că atunci când este necesară o logică de priorităţi, se utilizează construcţia **if – else – if**.

Pe de altă parte dacă nu se doreşte implementarea unei logici de priorităţi, ştiind 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. CONSTRUCŢII/INSTRUCŢIUNI DE CAZ

Construcţiile de caz compară o expresie cu o serie de cazuri şi execută instrucţiunea sau grupul de instrucţiuni asociat cu prima potrivire de caz. În cazul grupului de instrucţiuni multiple se utilizează **begin – end**.

Sintaxa instrucţiunilor de caz este următoarea:

Sintaxa
Instrucţiune case
case (legătură_*sau_registru_sau_simbol literal*) // compară legătura, registrul sau *potrivire_caz1*: *instrucţiune sau grup de instrucţiuni* // valoarea simbolului literal cu fiecare caz şi *potrivire_caz2*, // execută instrucţiunea sau grupul de instrucţiuni *potrivire_caz3*: *instrucţiune sau grup de instrucţiuni* // asociat cu prima potrivire de caz default: *instrucţiune sau grup de instrucţiuni* // execută predefinit (opţional) dacă **endcase** // nici un caz nu se potriveşte
Instrucţiune casez
casez (legătură_*sau_registru_sau_simbol literal*) // tratează z ca “nu contează”
Instrucţiune casex
casex (legătură_*sau_registru_sau_simbol literal*) // tratează x şi z ca “nu contează”

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

Dacă la mai multe cazuri se execută aceeaşi instrucţiune sau grup de instrucţiuni 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&apos;bxx,2'bx0,2'bx1,2'b0x,2&apos;b1x,
   2&apos;bzz,2'bz0,2'bz1,2'b0z,2&apos;b1z : $display("Eroare de selectie");
endcase

endmodule
```

2.4.5. CONSTRUCŢII/INSTRUCŢIUNI PENTRU REPETARE

Construcţiile de repetare apar numai în interiorul blocurilor procedurale. **Verilog** are patru astfel de construcţii (**forev**er, **repeat**, **while** , **for**) şi vor fi prezentate în continuare.

Forever

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

Sintaxa este următoarea:

forever *instrucţiune sau grup de instrucţiuni*

Trebuie acordată o atenţie deosebită la utilizarea acestei instrucţiuni: simularea se poate agăţa daca nu este prezent un constructor de sincronizare.

Repeat

Bucla **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) *instrucţiune sau grup de instrucţiuni*

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

While

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

Sintaxa este următoarea:

while (expresie) *instrucţiune sau grup de instrucţiuni*

For

Sintaxa este următoarea:

for (*atribuire_iniţială*; *expresie*; *atribuire_pas*) *instrucţiune sau grup de instrucţiuni*

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

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

Exemplu:

...

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

...

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

2.4.6. CONSTRUCŢII/INSTRUCŢIUNI DE DEZACTIVARE

Sintaxa este următoarea:

disable *nume_grup*;

Întrerupe execuţia unui grup denumit de instrucţiuni. Simularea acestui grup sare la sfârşitul grupului fără execuţia oricărui eveniment programat.

2.4.7. ATRIBUIREA CONTINUĂ; PROPAGAREA ÎNTÂRZIERILOR

Construcţiile de atribuiri continue manipulează legături (tipuri de date fire). Ele reprezintă conexiuni structurale.

Sintaxa este următoarea:

Sintaxa
Atribuire continuă explicită **tip_legătură** [*dimensiune*] *nume_legătură*; **assign** (“strength”) **#**(întârziere) *nume_legătură* = *expresie*;
Atribuire continuă implicită **tip_legătură** [*dimensiune*] (“strength”) *nume_legătură* = *expresie*;

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

- Tip_legătură poate fi orice tip de date legătură cu excepţia **trireg**.
- "Strength" (opţional) poate fi specificat numai când atribuirea continuă este combinată cu o declarare de legătură. Valoarea predefinită pentru "strength" este (strong1, strong0).
- Întârzierea (opţional) urmăreşte aceeaşi sintaxă ca la instanţierea primitivelor. Valoarea predefinită este zero.
- Expresia poate include orice tip de date, orice operator şi apelări de funcţii.
- Atribuirile continue modelează tampoane ("buffers") cu impedanţă înaltă şi pot modela logica de tip combinaţional. De fiecare dată când un semnal se schimbă pe partea din dreaptă, partea dreaptă este reevaluată şi rezultatul este atribuit legăturii din partea stângă.
- Atribuirile continue sunt declarate ăn afara blocurilor procedurale şi se extind peste orice atribuiri procedurale. Ele devin automat active la momentul zero şi sunt evaluate concurent cu blocurile procedurale, instanţierile de module şi 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 întîrzierilor

Atribuirile continue pot avea o întârziere specificată şi numai pentru toate tranziţiile. Poate fi specificat şi 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 execuţia 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, atrunci toate blocurile (**always** şi **initial**) vor începe să se execute de la momentul zero şi vor continua să se execute concurent; deseori acest lucru poate conduce la condiţii neprecizate şi ieşirile nu pot fi stabilite.

Controlul procedural al logicii combinaţionale

Pentru a modela logica de tip combinaţional, un bloc procedural trebuie să fie senzitiv la orice schimbare a intrărilor. Daca se utilizează instrucţiunea condiţională atunci este nevoie să se utilizeze şi partea “else”; lipsa acestei părţi schimbă logica într-un “latch”. Daca nu se doreşte utilizarea “else” atunci trebuie să se iniţializeze toate variabilele din blocul combinaţional chiar de la început.

Controlul procedural al logicii secvenţiale

Pentru a modela logica de tip combinaţional, un bloc procedural trebuie să fie senzitiv la frontul pozitiv sau negativ al semnalului de ceas. Dacă se doreşte o iniţializare asincronă atunci blocul trebuie să fie senzitiv şi la frontul semnalelor de “reset”, “preset”, “clear”, etc. Toate atribuirile la logica sevenţială 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 funcţionarea reală; nu este bine la sinteză să se utilizeze semnalul de ceas pentru a activa bistabile.

Blocuri numite

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

2.4.9. BLOCURI SPECIFICE

Au următoarea sintaxă:

Sintaxa
specify declaraţii_**specparam** verificări_constrângeri_timing întârziere_cale_simplă_pin-la-pin întârziere_cale_front-senzitiv_pin-la-pin întârziere_cale_stare-dependentă_pin-la-pin **endspecify**

Declaraţii_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 şiruri.

Verificări constrângeri de "timing"

Sunt task-uri sistem care modelează restricţiile la schimbările pe intrări, cum sunt timpii de "setup" şi de "hold".

Au sintaxa următoare:

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

Caracteristici:

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

<u>Întârzieri cale pin-la-pin</u>

- Întârziere cale simplă:
  (*port_intrare* polaritate:indicator_cale *port_ieţire*) = (întârziere);

- Întârziere cale senzitivă pe front:
  (front *port_intrare* indicator_cale (*port_ieţire* polaritate:sursă)) = (întârziere);
  - Front (opţional) poate fi *posedge* sau *negedge*. Dcaă nu este specificat sunt utilizate toate tranziţiile de intrare.
  - Sursa (opţional) este portul de intrare sau valoarea de ieşire 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_condiţie) *întârziere_cale_simplă_sau_front-senzitiv*
  if (următoarea_condiţie) *întârziere_cale_simplă_sau_front-senzitiv*
  ifnone *întârziere_cale_simplă*
  - Permite întârzieri diferite pentru aceeaşi cale care urmează a fi specificată, pe baza altor condiţii de intrare.
  - Condiţia poate fi bazată numai pe porturi de intrare.

- Cu condiţii pot fi utilizaţi majoritatea operatorilor, dar trebuie sa se rezume la adevărat/fals (o valoare logică “x” sau “z” este considerată adevărată; daca o condiţie se rezumă la un vector este utilizat numai *lsb*).
- Fiecare întârziere dependent de stare pentru aceeaşi cale trebuie să aibă o condiţie diferită sau un front-senzitiv diferit.
- Condiţia **ifnone** (opţional) poate fi numai o întârziere de cale simplă şi serveşte ca valoare predefinită dacă nu este evaluată nici o condiţie ca adevărată.

- Polaritatea (opţional) este ori + sau P şi 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 ieşire. Întârzierea de cale conectată paralel indică faptul că fiecare bit de intrare este conectat la bitul său corespondent de ieşire (bitul 0 la bitul 0, bitul 1 la bitul 1, ...).
- Întârzierea reprezintă faptul că pot fi specificate 1, 2, 3, 6 sau 12 tranziţii. Fiecare tranziţie poate avea o întârziere singulară sau un domeniu de întârziere minim:tipic:maxim.

Întârzieri	**Tranziţii reprezentate (în ordine)**
1	Toate tranziţiile de ieşire
2	Tranziţiile de ieşire crescătoare şi descrescătoare
3	Tranziţiile de ieşire crescătoare, descrescătoare şi “turn-off”
6	Tranziţii crescătoare, descrescătoare, 0->Z, Z->1, 1->Z, Z->0
12	Tranziţii 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 senzitive pe front sau prin controlul evenimentelor senzitive pe nivel.

Controlul întârzierii

Sintaxa este următoarea: ***#****întârziere*

Întârzie execuţia unei construcţii 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 ...) instrucţiune;*

Îtîrzie execuţia până ce apare o tranziţie logică pe un semnal. Frontul (opţional) poate fi *posedge* sau *negedge;* daca nu se specifică nici un front atunci este utilizată orice tranziţie logică. Semnalul poate fi scalar sau vector şi de orice tip de date.

Controlul evenimentelor senzitive pe nivel – construcţii wait

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

Întârzie execuţia 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 atribuiri 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 atribuiri blocante întârziate intern. Expresia este evaluată în pasul de timp în care este întâlnită instrucţiunea şi 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 instrucţiunea şi este atrbuită la sfârşitul pasului de timp specificat prin controlul sincronizărilor anterioare. Sincronizarea modelează transportul întârzierii.

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

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

2.5. TASK-URI ŞI FUNCŢII

2.5.1. TASK-URI

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

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

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

Task_urile pot avea orice număr de intrări, ieşiri sau “inout”-uri şi pot conţine sincronizări de tip **#**, **@** sau **wait**.

Task-urile sunt definite în modulul în care sunt utilizate. Este posibil să se definească un task într-un fişier separat şi să se utilizeze directiva de compilare “include” pentru a include task-ul în fişierul care instanţiază task-ul.

Sintaxa de definire a unui task este următoarea:

Sintaxa
task nume_*task*; declaraţii ***input, output****, şi* ***inout*** *declaraţii variabile locale* *construcţii_procedurale sau grup_instrucţiuni* **endtask**

Caracteristici:

- Variabilele locale declarate într-un task sunt locale acelui task.
- Ordinea de declarare dintr-un task defineşte cum sunt utilizate variabilele trimise task-ului de către apelanţi.
- Când nu sunt utilizate variabile locale, task-urile pot manipula variabile globale (în aces caz variabilele sunt declarate ca registre înainte de definirea task-ului, în modulul în care se defineşte task-ul). Când se utilizează variabile locale, ieşirea este asignată unei variabile regidtru numai la sfârşitul execuţiei task-ului.
- Task-urile pot apela alte task-uri sau funcţii.
- Task-urile pot fi utilizate pentru a modela atât logica combinaţională, cât şi logica secvenţială.
- Un task trebuie să fie apelat specificcu o instrucţiune. El nu poate fi utilizat cu o expresie aşa cum poate fi utilizată o funcţie.

Exemplu:

Fie un task definit în fişierul **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. FUNCŢII

O funcţie **Verilog** este asemănătoare cu un task, cu foarte puţine diferenţe, cum ar fi faptul că funcţia nu poate conţine decât o ieşire şi nu poate conţine întârzieri. Funcţiile reîntorc ca ieşire valoarea care este atribuită numelui funcţiei. O funcţie începe cu cuvântul cheie **function** şi se termină cu cuvântul cheie **endfunction**.

Funcţiile sunt definite în modulul în care sunt utilizate. Este posibil să se definească funcţii în fişiere separate şi să se utilizeze directiva de compilare "include" pentru a include funcţia în fişierul care instanţiază funcţia.

Sintaxa de definire a unei funcţii este următoarea:

Sintaxa
function [dimensiune_sau_tip] *nume_funcţie*; *declaraţii* ***input*** *declaraţii variabile locale* *construcţii_procedurale sau grup_instrucţiuni* **endfunction**

Caracteristici:

- Funcţiile trebuie să aibă cel puţin o intrare şi nu pot avea ieşiri sau "inout"-uri.
- Ele nu pot conţine sincronizări de tip **posedge**, **negedge**, **#**, **@** sau **wait**, ceea ce înseamnă că funcţiile se execută cu întârzierea zero.
- Dimensiune_sau_tip (opţional) este domeniul de biţi reîntors ca *[msb : lsb]*, sau cuvântul cheie **integer** sau **real**.
- Variabilele declarate într-o funcţie sunt locale acelei funcşii. Ordinea de declarare dintr-o funcţie defineşte cum sunt utilizate variabilele trimise funcţiei de către apelanţi.
- Când nu sunt utilizate variabile locale, funcţiile pot manipula variabile globale (în aces caz variabilele sunt declarate înainte de definirea funcţiei, în modulul în care se defineşte funcţia). Când se utilizează variabile locale, ieşirea este asignată unei variabile fir ("**wire**") numai la sfârşitul execuţiei funcţiei.
- Funcţiile pot apela alte funcţii, nu însă şi task-uri.
- Funcţiile pot fi utilizate pentru a modela doar logica combinaţională.

Exemplu:

Fie o funcţie definită în fişierul **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 [15:0] produs;
`include "functia_mea.v"

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

endmodule
```

2.6. MODELAREA MEMORIILOR ŞI A MAŞINILOR 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 conţine 1024 locaţii (adrese) de cuvinte de 8 biţi,

// 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];”.*

- **Iniţializarea memoriilor**

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

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

Adresele şi extensia fişierului sunt opţionale.

Exemplu de fişier:

```
//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 secţiunea programelor de test.

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

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

Există două tipuri de diagrame: Moore, unde ieşirile sunt funcţie numai de starea curentă şi Mealy, unde una sau mai multe ieşiri sunt funcţie de starea curentă şi 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 şi complexitatea porţilor pentru diagramele de stări.

Este bine de ţinut minte un lucru important când se codifică FSM şi anume că logica de tip combinaţional şi cea de tip secvenţial 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 uşor de citit şi de gestionat.

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

- Stilul codificării: Cel mai utilzat este binar şi complet decodificat.
- Partea combinaţională: Această secţiune poate fi modelată utilizând funcţii, construcţii de atribuire sau blocuri **always** cu o construcţie **case**.
- Partea secvenţială: Această secţiune trebuie să fie modelată utilizând logica senzitivă pe front cum sunt blocurile **alwazs** 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 şi scrierea codului RTL însuşi. În zilele actuale circuitele ASIC devin din ce în ce mai complexe şi 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 specificaţiile de proiectare ale "design under test" (DUT – proiectare în vederea testării). Specificaţile trebuie înţelese clar şi este necesar să fie realizat în detaliu un plan de test cu o documentare temeinică a arhitecturii programului de test şi a scenariilor de test. Primul pas în scrierea unui program de test este construirea unui "template" al lui, care va conţine registre (**reg**) pentru intrări, fire (**wire**) pentru ieşiri apoi instanţiază proiectul.

Structura unui "template" este următoarea:

```
module nume_testbench;
  reg [msb_r:0] intrări;
  wire [msb_w:0] ieşiri;

  nume_modul instanţă (
  .semnal1_modul (semnal1_instanţă),
  .semnal2_modul (semnal2_instanţă),
  …
  );

endmodule
```

Următorul pas este să se adauge logica de ceas. Înainte însă trebuie să se iniţializeze 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 utliza *`define* pentru a defini şi 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 opţiuni de compilare şi simulare, incluse în blocuri **initial**.

Astfel de opţiuni ar fi:

- *$dumpfile*: este utilizat pentru specificarea fişierului pe care îl utilizează simulatorul pentru a memora formele de undă.
- *$dumpvars*: instruieşte compiulatorul **Verilog** să înceapă cu semnalele dintr-un fişier.
- *$display*: este utilizat pentru a tipări pe ecran texte sau variabile.
- *$monitor*: păstrează urma modificărilor variabilelor din listă; cum apare o schimbare ea va fi afişată în format specific.
- *$finish*: este utilizat pentru terminarea simulării dupa un număr de unităţi de timp
- *$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** şi apoi specificarea tranziţiilor 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 apariţia căruia logica de reset va activa semnalul de *reset* pe frontul negativ de ceas şi 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 porneşte cu *reset* dezactivat, apoi se activează *reset* pentru câteva perioade de ceas şi apoi se dezactivează
- Enable test: activare/dezactivare *enable* după aplicarea *reset*
- Activare/dezactivare aleatoare pentru *enable* şi *reset*

Cazurile de test nu pot exista în acelaşi fişier, aşa că ele se codifică separat şi sunt incluse apoi în programul de test utilizând directiva **`include**. Se poate condiţiona terminarea simulării de un **event** *"terminate_sim".*

Se poate adăuga acum o logică de comparare a ceea ce se aşteaptă să se obţină la simulare şi ceea ce se obţine 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 şi variabile
- A nu se mixa elemente senzitive pe nivel şi pe front în acelaşi bloc
- A se evita mixarea de bistabile eşantionate pe front pozitiv şi pe front negativ
- Utilizarea parantezelor în vederea optimizării structurii logice
- Utilizarea instrucţiunilor de atribuire continuă pentru logica simplă combinaţională
- Utilizarea atribuirilor nonblocante pentru logica secvenţială şi a celor blocante pentru logice de tip combinaţional
- A nu se mixa atribuiri blocante şi nonblocante în acelaşi bloc **always**
- Atenţie la atrbuiri multiple la aceeaşi variabilă
- A se defini în mod explicit construcţiile **if-else** şi **case**

În sinteza logică nu sunt suportaţi toţi constructorii. Aceştia sunt următorii:

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

Sinteza logica nu suportă nici operatorii === şi !==.

Furnizorii de tool-uri şi utilizatorii de tool-uri pot defini task-uri şi funcţii specifice tool-ului lor, cum ar fi ieşiri de text sau afişare forme de undă. Task-urile sistem şi funcţiile încep cu simbolul"$".

Utilizatorii pot defini task-uri şi funcţii predefinite (“built-in”) utilizând interfaţa **Verilog** PLI (“Programming Lnaguage 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. Madhaven and W. Stapleton, *Digital Design and Synthesis with Verilog HDL*, published by Automata Publishing Co., Cupertino, CA, 1993, ISBN 0-9627488-2-X, $65.

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