CAPITOLUL 2

SIMULAREA DESCRIERILOR VHDL

În prima parte a acestui capitol se prezintă principiul simulatoarelor și al simulării asistate de calculator pentru sistemele digitale. În continuare, se descriu tipurile de date și instrucțiunile VHDL care sunt utile pentru simulare, se introduc noțiunile care sunt importante pentru scrierea corectă a modelelor destinate simulării și se prezintă detalii despre scrierea bancurilor de test. În ultima parte a capitolului se prezintă etapele necesare pentru simularea unui proiect simplu utilizând simulatorul mediului de proiectare Xilinx Vivado Design Suite.

2.1 Simularea asistată de calculator

2.1.1 Principiul simulării asistate de calculator

Simularea unui sistem digital permite verificarea funcționării sistemului pe baza specificării acestuia prin oricare din metodele uzuale (limbaje de descriere hardware, scheme, diagrame de stare), înainte de implementarea efectivă. Simularea asistată de calculator este metoda cea mai utilizată pentru simularea sistemelor digitale. Utilizarea acestei metode a devenit posibilă pentru sistemele digitale complexe odată cu creșterea semnificativă a performanței sistemelor de calcul. Simularea se realizează cu ajutorul unui program simulator, care construiește un model al sistemului pe baza descrierii acestuia de către proiectant, model care este echivalent cu o copie virtuală a sistemului proiectat. Simulatorul execută apoi modelul sistemului și analizează răspunsul acestuia la o serie de combinații ale intrărilor aplicate sistemului într-un anumit interval de timp; o asemenea combinație a intrărilor este numită *vector de test* sau *stimul*.

Pe lângă vectorii de test, proiectantul poate specifica și ieșirile așteptate ale sistemului pentru fiecare vector. În acest caz, simulatorul va compara ieșirile generate prin execuția modelului cu cele specificate de proiectant, care sunt cunoscute ca fiind corecte, afișând mesaje de eroare în cazul existenței unor diferențe între acestea. Dacă se detectează erori la simulare, descrierea sistemului poate fi corectată mult mai simplu decât atunci când sistemul ar fi fost deja implementat.

Pentru simularea sistematică a funcționării unui sistem, de multe ori se creează un *banc de test ("test bench"*) constând din sistemul care trebuie testat și module adiționale pentru generarea vectorilor de test și aplicarea acestora la intrările sistemului. Vectorii de test pot fi reprezentați fie de toate combinațiile posibile ale intrărilor sistemului (dacă este posibil), fie de combinațiile reprezentative pentru toate situațiile de funcționare. Prin utilizarea bancurilor de test, este posibilă simularea sistemului în condiții reale, când acestuia i se aplică intrări din mediul exterior de la un operator uman sau de la un alt circuit sau sistem digital.

Simularea asistată de calculator are următoarele avantaje principale:

- Simularea permite experimentarea cu diferite variante de proiectare şi condiții de funcționare ale sistemului, fără a fi necesară implementarea acestor variante. Proiectantul va putea implementa apoi varianta cea mai eficientă din punctul de vedere al performanțelor şi al costului.
- Prin utilizarea simulării asistate de calculator este posibilă testarea sistematică a sistemelor prin aplicarea unui număr mare de vectori de test la intrările sistemului, vectori care pot fi generați prin program sau pot fi specificați sub formă tabelară.
- Pentru circuite sau sisteme digitale complexe, simularea asigură reducerea costurilor și reducerea numărului erorilor de proiectare comparativ cu cazul în care se utilizează un prototip hardware pentru testare. Realizarea unor circuite complexe cum sunt microprocesoarele moderne nu ar fi fost posibilă fără utilizarea intensivă a simulării asistate de calculator.

2.1.2 Tipuri de simulare

Simularea poate fi executată în diferite etape ale procesului de proiectare și poate utiliza diferite nivele de abstractizare ale sistemului specificat printr-o anumită metodă. Principalele tipuri de simulare sunt prezentate în continuare.

Simularea funcțională constă în simularea unei descrieri la nivel înalt a sistemului, descriere care specifică funcționarea sistemului și nu structura acestuia. Pentru descrierea sistemului se pot utiliza diferite limbaje de modelare, cum sunt limbajele de programare. Simularea constă atunci în compilarea și execuția modelului respectiv. În multe cazuri, simularea funcțională se bazează pe un model al sistemului sub forma unei descrieri într-un limbaj de descriere hardware HDL (*Hardware Description Language*).

Simularea logică reprezintă analiza funcționării sistemului pe baza valorii unor variabile logice. Această simulare se numește și simulare *la nivelul transferurilor*

între registre (RTL – *Register Transfer Level*), deoarece variabilele simulate sunt cele păstrate în registre. Simularea evaluează în fiecare ciclu de ceas funcțiile logice ale căror valori sunt înscrise în registre.

Simularea la nivel de circuite se referă la analiza funcționării unor modele ale circuitelor reprezentate prin interconectarea unor dispozitive electronice, cum sunt tranzistoare, rezistențe și condensatoare. Această simulare constă în calcularea nivelelor de tensiune în funcție de timp din toate nodurile circuitului sau dintr-o parte a nodurilor. Aceasta presupune formularea și rezolvarea unui mare număr de ecuații diferențiale, necesitățile de memorie și de timp de calcul fiind ridicate.

Simularea temporală constă în simularea funcționării sistemului după determinarea întârzierii reale a semnalelor pe diferitele căi de date. Această simulare se utilizează în special pentru circuitele programabile, la care întârzierile semnalelor nu vor fi cunoscute decât în urma implementării sistemului, atunci când se poate determina numărul de conexiuni programabile utilizate pentru rutarea semnalelor. Informațiile despre întârzierile semnalelor sunt furnizate simulatorului printr-o operație numită adnotare inversă. Simularea temporală permite analiza funcționării în condiții reale a sistemului și determinarea frecvenței maxime de funcționare a acestuia.

2.1.3 Funcționarea unui simulator

Un program de simulare necesită două tipuri de intrări: un fișier cu descrierea sistemului care trebuie simulat și un set de stimuli pentru definirea tuturor semnalelor de intrare pe durata timpului de simulare. Sistemul poate fi descris printr-o listă de conexiuni sau printr-un limbaj de descriere hardware. În cazul în care sistemul este specificat prin scheme sau diagrame de stare, mai întâi trebuie creată o descriere echivalentă a sistemului sub forma unei liste de conexiuni. Această descriere poate fi într-un format specific listelor de conexiuni, cum este formatul EDIF (*Electronic Design Interchange Format*), sau poate fi o descriere structurală a sistemului într-un limbaj de descriere hardware.

Proiectantul trebuie să specifice setul de stimuli care va fi aplicat la intrările sistemului pe durata simulării acestuia. Opțional, proiectantul poate specifica și semnalele de ieșire care trebuie generate de sistem pentru fiecare stimul. Pentru definirea stimulilor, mediile de proiectare sau simulatoarele pun la dispoziție diferite interfețe, cele mai utilizate fiind interfețele grafice, interfețele bazate pe fișiere text și cele la nivelul liniei de comandă. Interfețele grafice permit specificarea valorii semnalelor aplicate la intrările sistemului sub forma unor diagrame de timp. Aceste interfețe sunt utile atunci când trebuie definit un număr redus de intrări. Atunci când există un număr mai mare de semnale de intrare, sau trebuie definit un număr mare de tranziții ale acestor semnale, este mai utilă specificarea într-un fișier text a unor comenzi care definesc valorile semnalelor de intrare și momentele de timp în care semnalele își modifică valoarea. Pot exista și comenzi care specifică execuția simulării pentru un anumit timp. Introducerea comenzilor de simulare prin linia de comandă este utilă atunci când trebuie modificate valorile unor semnale care au fost definite anterior prin alte metode.

2.2 Tipuri de date VHDL pentru simulare

Următoarele tipuri de date se pot utiliza exclusiv pentru simulare: tipuri flotante, tipuri fizice, tipuri de acces și tipul fișier. Tipurile flotante permit definirea unor numere reale. Tipurile fizice permit specificarea unor cantități fizice, cum este timpul. Tipurile de acces sunt similare cu pointerii din limbajele de programare. Aceste tipuri permit alocarea dinamică a memoriei pentru un obiect și eliberarea memoriei. Tipul fișier permite declararea unui obiect fișier, care conține o secvență de valori de un anumit tip. Accesul la obiectele dintr-un fișier se realizează prin proceduri și funcții speciale.

2.2.1 Tipuri standard utilizate pentru simulare

Un număr de tipuri sunt predefinite în limbajul VHDL. Aceste tipuri sunt definite într-un pachet numit STANDARD.

Tabelul 2.1 prezintă tipuri de date care sunt specifice pentru simulare și care sunt definite în pachetul STANDARD. De remarcat că se pot utiliza pentru simulare și celelalte tipuri definite în pachetul STANDARD, cum sunt **BIT**, **BOOLEAN**, **INTEGER** sau **BIT_VECTOR**. Tipurile **CHARACTER** și **STRING** se pot utiliza și pentru sinteză. Tipurile FILE_OPEN_KIND, FILE_OPEN_STATUS și subtipul DELAY_LENGTH nu sunt definite în versiunea inițială a standardului IEEE 1076-1987 pentru limbajul VHDL (VHDL-87), ci numai începând cu versiunea VHDL-93 a limbajului.

Тір	Categorie
CHARACTER	Tip enumerat
SEVERITY_LEVEL	Tip enumerat
FILE_OPEN_KIND	Tip enumerat
FILE_OPEN_STATUS	Tip enumerat
REAL	Tip flotant
TIME	Tip fizic
DELAY_LENGTH	Subtip al tipului TIME
STRING	Tablou cu tipul CHARACTER

Tabelul 2.1. Tipuri de date specifice pentru simulare care sunt definite în pachetul STANDARD.

2.2.1.1 Tipul CHARACTER

Tipul **CHARACTER** este un tip enumerat definit în pachetul STANDARD. În versiunea VHDL-87 a limbajului, valorile de acest tip sunt reprezentate de cele 128 de caractere ale setului ASCII de 7 biți. Începând cu versiunea VHDL-93, aceste valori sunt reprezentate de cele 256 de caractere ale setului ISO 8859-1 (Latin 1). Caracterele alfanumerice și cele speciale sunt indicate prin includerea lor între ghilimele simple, de exemplu, 'a'. În cazul tipului **CHARACTER**, literele mari sunt diferite de literele mici. Uneori, pentru evitarea ambiguităților, trebuie să se utilizeze o expresie calificată pentru specificarea explicită a faptului că un literal este de tip **CHARACTER**:

```
CHARACTER'('1')
```

Caracterele de control, având codurile cuprinse între x"00" și x"1F", sunt specificate simbolic, ca în exemplele următoare:

NUL (Null)	x"00"
STX (Start of Text)	x"02"
ETX (End of Text)	x"03"
BEL (Bell)	x"07"
BS (Backspace)	x"08"
HT (Horizontal Tabulation)	x"09"
LF (Line Feed)	x"0A'
CR (Carriage Return)	x"0D'
ESC (<i>Escape</i>)	x"1B"
· · ·	

Operatorii care se pot utiliza cu tipul CHARACTER sunt operatorii relaționali: =, /=, <, <=, >, >=.

2.2.1.2 Tipul SEVERITY_LEVEL

Tipul SEVERITY_LEVEL este un tip enumerat definit în pachetul STANDARD, utilizat în clauza severity a unei instrucțiuni assert. Această instrucțiune este utilă pentru raportarea unor mesaje despre funcționarea unui model în timpul simulării, în funcție de anumite condiții exprimate sub forma unor expresii booleene. Clauza severity permite specificarea unui nivel de severitate care clasifică mesajele raportate în patru categorii. Instrucțiunea assert va fi prezentată în secțiunea 2.3.3.

Tipul SEVERITY_LEVEL constă din patru valori, fiecare indicând câte o categorie a mesajului raportat proiectantului de instrucțiunea assert: NOTE, WARNING, ERROR și FAILURE. Mesajele din categoria NOTE sunt utile pentru informarea proiectantului despre modul de desfășurare a procesului de simulare sau despre etapa în care se află acest proces. Mesajele din categoria WARNING pot fi utilizate pentru atenționarea proiectantului despre condiții care ar putea indica un comportament diferit de cel așteptat. Mesajele din categoria ERROR sunt utilizate pentru indicarea unor condiții care determină o funcționare incorectă a modelului, iar mesajele din categoria FAILURE sunt utilizate pentru indicarea unor condiții din cadrul modelului care pot avea efecte dezastruoase, cum este o operație de împărțire la zero.

Declarația acestui tip este următoarea:

type SEVERITY LEVEL is (NOTE, WARNING, ERROR, FAILURE);

Operatorii care sunt predefiniți pentru tipul SEVERITY_LEVEL sunt operatorii relaționali: =, /=, <, <=, >, >=.

2.2.1.3 Tipurile FILE_OPEN_KIND și FILE_OPEN_STATUS

Tipurile FILE_OPEN_KIND și FILE_OPEN_STATUS se utilizează la simulare pentru deschiderea fișierelor destinate testării modelelor prin bancuri de test.

Tipul FILE_OPEN_KIND este un tip enumerat constând din trei valori care pot fi specificate de parametrul open_kind al procedurii FILE_OPEN, procedură utilizată pentru deschiderea unui fișier. Valorile enumerate specifică modul de acces la fișier. Declarația acestui tip este următoarea:

```
type FILE_OPEN_KIND is (
READ_MODE,
WRITE_MODE,
APPEND_MODE);
```

Semnificația celor trei valori este următoarea:

READ_MODE	Mod de acces pentru citire.
WRITE_MODE	Mod de acces pentru scriere.
APPEND_MODE	Mod de acces pentru scriere; informația este adăugată la sfârșitul fișierului existent.

Tipul FILE_OPEN_STATUS este un tip enumerat constând din patru valori care pot fi returnate în parametrul file_status al procedurii FILE_OPEN care se utilizează pentru deschiderea unui fișier. Aceste valori indică rezultatul procedurii FILE OPEN. Declarația acestui tip este următoarea:

Semnificația acestor valori este următoarea:

OPEN_OK	Apelul procedurii FILE_OPEN s-a realizat cu succes.
STATUS_ERROR	Obiectul de tip fișier este asociat deja cu un fișier extern.
NAME_ERROR	La încercarea de citire dintr-un fișier extern: fișierul nu există; la încercarea de scriere într-un fișier extern: fișierul nu poate fi creat.
MODE_ERROR	Fișierul extern nu poate fi deschis cu tipul de acces specificat de parametrul open_kind al procedurii FILE_OPEN.

Operatorii care sunt predefiniți pentru acest tip sunt operatorii relaționali: =, /=, <, <=, >, >=.

2.2.1.4 Tipuri flotante

Valorile flotante se utilizează pentru aproximarea numerelor reale. La fel ca și tipurile întregi, tipurile flotante pot fi restrânse și ele într-un domeniu. Singurul tip flotant predefinit al limbajului VHDL este tipul **REAL**, care cuprinde cel puțin domeniul [-1.0 E38, +1.0 E38], în funcție de implementare. Valorile reale trebuie scrise cu punctul zecimal, ca în exemplele următoare:

Operatorii definiți pentru tipurile flotante sunt operatorii relaționali (=, /=, <, <=, >, >=) și operatorii aritmetici, cu excepția operatorilor mod și rem: semnul +, semnul -, abs, +, -, *, /, **.

2.2.1.5 Tipuri fizice

Tipurile fizice specifică nu numai valori ale obiectelor, ci și unitățile de măsură în care se exprimă acele valori. Aceasta permite specificarea cu precizie a unor cantități fizice ca timpul, tensiunea, curentul, capacitatea, distanța, temperatura etc.

Declarația unui tip fizic cuprinde specificarea unui domeniu al valorilor posibile pentru unitățile de măsură ale tipului și specificarea unităților de măsură. Domeniul este specificat prin valorile limită, care pot fi valori întregi sau flotante (de tip **REAL**). În declarația unui tip fizic se pot specifica două feluri de unități de măsură: o unitate *primară* și una sau mai multe unități *secundare*, acestea din urmă fiind definite ca multipli întregi ai unității primare. Specificarea unității primare este obligatorie, în timp ce specificarea unităților secundare este opțională.

Observație

Domeniul specificat în declarația tipului se referă numai la unitatea de măsură primară.

Singurul tip fizic predefinit al limbajului VHDL este tipul TIME, care este definit în pachetul STANDARD. Domeniul acestui tip cuprinde cel puțin domeniul valorilor întregi. Unitatea primară a tipului TIME este fs (femtosecunde). În cazul unei implementări pe 32 de biți, acest tip este definit astfel:

```
type TIME is range -2147483647 to 2147483647
units
    fs;
    ps = 1000 fs;
    ns = 1000 ps;
    us = 1000 ns;
    ms = 1000 us;
    sec = 1000 ms;
    min = 60 sec;
    hr = 60 min;
end units;
```

Exemple de variabile de tip TIME sunt următoarele:

```
variable T1 : TIME := 3.5 ns;
variable T2 : TIME := 5 ns;
```

Pachetul STANDARD conține și o declarație a unui subtip al tipului TIME, numit DELAY_LENGTH. Acest subtip este utilizat pentru reprezentarea valorilor timpului de simulare. Declarația subtipului DELAY_LENGTH este următoarea:

subtype DELAY LENGTH is TIME range 0 fs to TIME 'high;

Atributul 'high indică valoarea maximă a domeniului tipului căruia i se aplică (în acest caz, TIME).

Operatorii definiți pentru tipurile fizice sunt operatorii relaționali (=, /=, <, <=, >, >=) și o parte a operatorilor aritmetici: semnul +, semnul -, abs, +, -, *, /.

Tipurile fizice pot fi utilizate pentru crearea bancurilor de test. Utilizatorul poate defini alte tipuri fizice bazate pe alte unități de măsură, cum sunt metri, grame, amperi etc.

2.2.1.6 Tabloul predefinit STRING

Tabloul **STRING** este de tip **CHARACTER** și reprezintă un șir de caractere. Definiția acestui tablou este următoarea:

type **STRING** is array (**positive** range <>) of **CHARACTER**;

Tabloul **STRING** este definit cu un domeniu nelimitat, ceea ce este indicat prin simbolul <>. Declarația indexului pentru tabloul **STRING** arată că indexul trebuie să fie un întreg pozitiv, astfel încât nu poate fi zero.

Pentru utilizarea unui semnal de acest tip, domeniul său trebuie restrâns în declarația de semnal, ca în exemplul următor:

```
signal Mes1 : STRING (1 to 40);
```

2.2.2 Fișiere

Fișierele conțin secvențe de valori cu un tip specificat. Fișierele se pot utiliza pentru citirea unor stimuli necesari simulării și pentru scrierea unor date. Un tip fișier permite definirea obiectelor reprezentând fișiere. Valoarea unui obiect de tip fișier este secvența valorilor conținute în fișierul fizic. Declarația unui tip fișier are forma următoare:

```
type FT is file of tip_bază;
```

Tipul $tip_{bază}$ din declarația tipului fișier definește subtipul valorilor conținute într-un fișier cu tipul declarat. Subtipul nu poate fi bazat pe un tip fișier sau pe un tip de acces. Dacă se utilizează un tablou, acesta trebuie să fie unidimensional. În liniile următoare se declară un tip fișier și un obiect de acest tip. type TEXT is file of STRING; file VectorFile : TEXT;

Tipul **TEXT** este declarat în pachetul TEXTIO, care este definit în standardul IEEE 1076.

Atunci când se declară un tip fișier, sunt definite în mod implicit mai multe operații cu obiectele de acest tip. Aceste operații cuprind deschiderea unui fișier, închiderea unui fișier, citirea dintr-un fișier, scrierea într-un fișier și testarea sfârșitului unui fișier. Pentru tipul FT declarat anterior, operațiile implicite sunt definite de procedurile și funcțiile următoare:

Procedurile FILE_OPEN și FILE_CLOSE sunt definite numai începând cu versiunea VHDL-93 a limbajului. Procedurile READ și WRITE, care au mai multe versiuni, pot fi utilizate după ce fișierul este deschis. Limbajul VHDL permite operații de citire și scriere cu fișiere având tipuri predefinite. Pentru alte tipuri, utilizatorul trebuie să scrie proceduri proprii sau să folosească proceduri puse la dispoziție de pachetele existente.

Exemplul 2.1 ilustrează declarația unui tip de fișier text, declarația a două fișiere de acest tip și utilizarea procedurilor FILE_OPEN și FILE_CLOSE pentru aceste fișiere.

Exemplul 2.1

```
-- Partea declarativă a arhitecturii
type TEXT_FILE_TYPE is file of STRING;
-- Partea descriptivă a arhitecturii
process
    file InFile : TEXT_FILE_TYPE;
    file OutFile : TEXT_FILE_TYPE;
    variable FileStatus : FILE_OPEN_STATUS;
begin
    FILE_OPEN (FileStatus, InFile, "input.txt", READ_MODE);
    FILE_OPEN (FileStatus, OutFile, "output.txt", WRITE_MODE);
    -- Aici se pot executa operații de citire și scriere
    FILE_CLOSE (InFile);
    FILE_CLOSE (OutFile);
    wait;
end process;
```

Începând cu versiunea VHDL-93 a limbajului, este posibilă deschiderea implicită a fișierului în declarația fișierului, după cum se ilustrează în continuare pentru cele două fișiere din Exemplul 2.1:

file InFile : TEXT_FILE_TYPE open READ_MODE is "input.txt"; file OutFile : TEXT_FILE_TYPE open WRITE_MODE is "output.txt";

2.3 Instrucțiuni VHDL pentru simulare

2.3.1 Instrucțiunea WAIT FOR

Instrucțiunea wait for permite suspendarea execuției unui proces pentru un timp specificat. Sintaxa instrucțiunii wait for este următoarea:

wait for expresie_de_timp;

De exemplu, următoarea instrucțiune suspendă execuția procesului în care este plasată pentru 10 ns:

wait for 10 ns;

2.3.2 Specificarea întârzierii inerțiale și a întârzierii de transport

În limbajul VHDL există două tipuri de întârzieri care pot fi utilizate pentru modelarea sistemelor reale, întârzierea inerțială și întârzierea de transport. Aceste întârzieri nu se pot utiliza pentru sinteza logică.

Întârzierea inerțială este implicită, fiind utilizată atunci când nu se specifică tipul întârzierii. Utilizarea clauzei after presupune, în mod automat, întârzierea inerțială. Într-un model cu întârziere inerțială, două modificări consecutive ale valorii unui semnal de intrare nu vor modifica valoarea unui semnal de ieșire dacă timpul dintre aceste modificări este mai scurt decât întârzierea specificată. Această întârziere reprezintă inerția circuitului real. Dacă, de exemplu, apar anumite impulsuri de durată scurtă ale semnalelor de intrare, semnalele de ieșire vor rămâne neschimbate.

Figura 2.1 ilustrează întârzierea inerțială cu ajutorul unui buffer. Bufferul cu o întârziere de 20 ns are o intrare A și o ieșire B. Semnalul A se modifică de la '0' la '1' la momentul 10 ns și de la '1' la '0' la momentul 20 ns. Impulsul semnalului de intrare are o durată de 10 ns, care este mai mică decât întârzierea introdusă de buffer. Ca urmare, semnalul de ieșire B va rămâne la valoarea '0'.

Bufferul din figura 2.1 poate fi modelat prin următoarea instrucțiune de asignare:

B <= A after 20 ns;

Întârzierea de transport trebuie specificată în mod explicit prin cuvântul cheie transport. Aceasta reprezintă întârzierea unei conexiuni, în care efectul apariției unui impuls al unui semnal de intrare este propagat la ieșire cu întârzierea specificată, indiferent de durata acestui impuls. Întârzierea de transport este utilă pentru modela-rea liniilor de transmisie și a conexiunilor dintre componente.



Figura 2.1. Ilustrarea întârzierii inerțiale.

Considerând același buffer din figura 2.1 și semnalul de intrare A cu aceeași formă, dacă se înlocuiește întârzierea inerțială cu cea de transport se obține semnalul de ieșire B din figura 2.2. Impulsul semnalului de intrare este propagat nemodificat la ieșire cu o întârziere de 20 ns.

Dacă se utilizează întârzierea de transport, bufferul din figura 2.2 poate fi modelat prin următoarea instrucțiune de asignare:



Figura 2.2. Ilustrarea întârzierii de transport.

2.3.3 Instrucțiunea ASSERT

Instrucțiunea assert este utilă pentru afișarea unor mesaje de avertisment sau de eroare în timpul simulării unui model. Această instrucțiune testează valoarea unei

condiții booleene și afișează mesajul specificat în cazul în care condiția este falsă. Sintaxa instrucțiunii este următoarea:

```
assert condiție
[report șir_de_caractere]
[severity nivel de severitate];
```

Condiția specificată este o expresie care trebuie să se evalueze la o valoare booleană. Dacă această valoare este TRUE, instrucțiunea nu are nici un efect. Dacă valoarea este FALSE, se afișează textul specificat în clauza report. Clauza opțională report poate avea ca argument un șir de caractere, cu tipul predefinit **STRING**. Dacă această clauză nu este specificată, șirul de caractere care va fi afișat în mod implicit va fi **"Assertion violation"**. Clauza opțională severity permite specificarea nivelului de severitate al violării aserțiunii. Nivelul de severitate trebuie să fie o expresie cu tipul SEVERITY_LEVEL. Acest tip este predefinit și conține următoarele valori, în ordinea crescătoare a nivelului de severitate: NOTE, WARNING, ERROR și FAILURE. Dacă această clauză este omisă, se va presupune nivelul de severitate implicit ERROR. Utilizarea nivelelor de severitate este descrisă pe scurt în continuare.

- Nivelul NOTE poate fi utilizat pentru afișarea unor informații despre modul de desfășurare al simulării.
- Nivelul WARNING poate fi utilizat în situațiile în care simularea poate fi continuată, dar este posibil ca rezultatele să fie imprevizibile.
- Nivelul ERROR se utilizează atunci când violarea aserțiunii reprezintă o eroare care determină funcționarea incorectă a modelului, în acest caz execuția simulării fiind oprită.
- Nivelul FAILURE se utilizează atunci când violarea aserțiunii reprezintă o eroare fatală, cum este împărțirea la zero sau adresarea unui tablou cu un index care depășește domeniul permis. Și în acest caz, execuția simulării va fi oprită.

În Exemplul 2.2, atunci când ambele semnale R și S au valoarea '1', se afișează mesajul specificat și simularea va fi oprită.

```
Exemplul 2.2
```

```
assert not (R = '1' and S = '1')
  report "Ambele semnale R si S au valoarea '1'"
  severity ERROR;
```

Pentru afișarea unui mesaj în mod necondiționat, se poate utiliza condiția FALSE, ca în exemplul următor:

assert (FALSE) report "Start simulare";

Pentru asemenea cazuri, începând cu versiunea VHDL-93 a limbajului, se poate utiliza clauza report ca o instrucțiune completă, fără clauza assert *condiție*.

Instrucțiunea assert descrisă anterior este o instrucțiune secvențială, presupunând că ea apare într-un proces sau subprogram. Există însă și o versiune concurentă a acestei instrucțiuni. Versiunea concurentă are un format identic cu versiunea secvențială, dar poate apare numai în afara unui proces sau subprogram. Instrucțiunea concurentă assert se execută ori de câte ori se modifică unul din semnalele din expresia condițională, spre deosebire de instrucțiunea secvențială assert, care se execută atunci când se ajunge la această instrucțiune într-un proces sau subprogram.

2.4 Modelarea pentru simulare

Limbajul VHDL a fost elaborat inițial ca un limbaj pentru simulare, fiind utilizat pentru sinteză doar mai târziu. Deși în prezentarea limbajului VHDL accentul se va pune pe utilizarea acestuia pentru sinteză, pentru înțelegerea limbajului este importantă cunoașterea modului în care un simulator VHDL execută instrucțiunile limbajului. Această cunoaștere poate clarifica semantica anumitor construcții VHDL, ceea ce va permite scrierea unor modele corecte.

Unele construcții ale limbajului VHDL sunt specifice pentru simulare și verificare, nu pentru sinteză. Utilitarele de sinteză pot ignora aceste construcții. Totuși, este de dorit ca rezultatele funcționale ale simulării codului să corespundă cu funcționarea circuitelor generate prin sinteză, în special dacă se vor folosi pentru același cod atât utilitare de simulare, cât și de sinteză. De aceea, proiectantul trebuie să asigure ca modelele create pentru utilitarele de sinteză să fie corecte și pentru utilitarele de simulare.

2.4.1 Simularea bazată pe evenimente

Toate simulatoarele VHDL sunt bazate pe evenimente. Un eveniment constă în modificarea stării unui semnal. Există trei concepte importante ale simulării bazate pe evenimente. Acestea sunt timpul de simulare, procesarea evenimentelor și întârzierea delta.

În timpul simulării, simulatorul ține evidența timpului de simulare, care este momentul de timp pentru care s-a realizat simularea și nu timpul necesar pentru simulare. Acest timp este măsurat ca un multiplu întreg al unei unități de timp care se numește limită de rezoluție. Simulatorul nu poate măsura intervale de timp mai mici decât această limită. Pentru simularea la nivel de porți logice sau la nivelul transferurilor între registre, limita de rezoluție poate fi, de exemplu, de o picosecundă (ps).

La modificarea stării unui semnal, se plasează un eveniment într-o coadă de evenimente pentru timpul de simulare la care a apărut această modificare. Atunci când simulatorul procesează evenimentul, acesta reevaluează fiecare instrucțiune a cărei intrare este semnalul care a determinat evenimentul (deci, instrucțiunile care sunt "sensibile" la acel semnal). Această reevaluare determină modificarea altor semnale și generarea altor evenimente. Considerăm procesul din Exemplul 2.3, care conține o singură instrucțiune de asignare.

Exemplul 2.3

```
proc1: process (A, B, C)
begin
        X <= A and B and C;
end process proc1;
```

La modificarea stării unuia din semnalele A, B sau C, instrucțiunea este reevaluată, rezultând o nouă valoare pentru semnalul X. Deoarece în această instrucțiune nu a fost utilizată o clauză after pentru a specifica o întârziere, se planifică un eveniment pentru semnalul X la timpul curent de simulare, eveniment care constă în modificarea valorii acestui semnal. Aceasta poate introduce probleme potențiale atunci când semnalul X trebuie actualizat în același timp cu unul din semnalele din care este generat. Pentru a rezolva această problemă, limbajul VHDL introduce noțiunea de *întârziere delta* (δ). Întârzierea delta poate fi considerată ca o întârziere infinitezimală (foarte mică) care implică execuția unui *ciclu delta* pentru actualizarea semnalelor (ciclul delta este explicat în secțiunea 2.4.3). Astfel, semantica instrucțiunii precedente de asignare este că valoarea expresiei din dreapta de la timpul curent de simulare, T_c, este planificată pentru a fi asignată la semnalul X după o întârziere delta de la timpul curent de simulare, deci la momentul T_{c+ δ}.

La momentul $T_{c+\delta}$ se procesează toate evenimentele planificate pentru acest moment, rezultând noi evenimente. Unele din acestea pot fi planificate pentru timpul curent de simulare, ceea ce înseamnă că ele sunt planificate pentru momentul $T_{c+2\delta}$. După o nouă întârziere delta, evenimentele sunt procesate din nou, rezultând noi evenimente, și așa mai departe, până când nu vor mai fi alte evenimente planificate pentru timpul curent de simulare. Doar atunci timpul de simulare este incrementat. Această funcționare a unui simulator bazat pe evenimente este ilustrată în secțiunile următoare.

2.4.2 Drivere de semnal

Considerăm din nou procesul din Exemplul 2.3. După cum s-a arătat anterior, dacă se modifică valoarea semnalelor A, B sau C, se planifică asignarea valorii funcției ȘI logic dintre semnalele A, B și C din momentul curent de simulare, T_c , la semnalul X după o întârziere delta, deci la momentul $T_{c+\delta}$. Cu alte cuvinte, se planifică un eveniment pentru *driverul semnalului* X. Un driver de semnal este reprezentat printr-o *formă de undă proiectată a ieșirii*. De fiecare dată când se efectuează o asignare la un semnal, se actualizează forma de undă proiectată a semnalului respectiv.

O formă de undă proiectată este un set de tranzacții care specifică noi valori pentru un semnal și momentele de timp în care se va realiza actualizarea semnalului. La simularea modelelor elaborate pentru sinteză, există, în principiu, două tranzacții care trebuie păstrate pentru fiecare semnal dat: tranzacția curentă, care specifică valoarea curentă și momentul actual de timp, și tranzacția următoare, dacă există, care specifică noua valoare a semnalului la următoarea întârziere delta. Totuși, aceasta nu înseamnă că vor fi necesare maxim două cicluri delta înaintea momentului următor de simulare, ceea ce se va ilustra la prezentarea simulării unui model cu mai multe instrucțiuni concurente.

Observație

Noțiunile de întârzieri delta, tranzacții, drivere de semnal și forme de undă proiectate sunt prezentate doar din punct de vedere conceptual. Diferitele simulatoare VHDL pot implementa aceste concepte în mod diferit față de descrierea prezentată. Simulatoarele VHDL trebuie să respecte doar specificațiile operaționale definite în manualul de referință al limbajului (standardul IEEE 1076), dar implementarea este specifică fiecărui sistem de proiectare.

Ca un exemplu, în urma execuției procesului proc1 rezultă driverul pentru semnalul X din figura 2.3.



Figura 2.3. Driver pentru semnalul X reprezentat printr-o formă de undă proiectată a ieșirii.

Dacă semnalele *B* și *C* au valoarea 'l' logic, iar valoarea semnalului *A* se modifică de la '0' la 'l' la momentul de timp de 5 ns, după cum se ilustrează în figura 2.4, atunci când timpul de simulare este de 5 ns, se va executa procesul proc1. În acest moment, valoarea curentă a semnalului *X* este '0' și se adaugă o tranzacție la driverul semnalului *X*, ('l', 5 ns + δ), unde δ este o întârziere infinitezimală. Atunci când timpul curent de simulare, T_c, trece de 5 ns, singura tranzacție rămasă pentru driverul semnalului *X* va fi ('l', T_c).

Considerăm procesul din Exemplul 2.4, în care există două asignări la același semnal.

Exemplul 2.4

```
proc2: process (A, B, C)
begin
    X <= '0';
    if (A = B or C = '1') then
        X <= '1';
    end if;
end process;</pre>
```

Dacă semnalele A, B și C au forma din figura 2.5, atunci când timpul curent de simulare este de 5 ns, tranziția semnalului B determină execuția procesului. Prima

asignare secvențială determină tranzacția ('0', 5 ns + δ). Deoarece valoarea curentă a semnalului X este '1', această tranzacție este adăugată la driverul semnalului X. Se evaluează apoi condiția pentru instrucțiunea *if*. Deoarece expresia este falsă, nu se execută alte instrucțiuni, iar procesul este suspendat.



Figura 2.4. Diagrama de timp și forma de undă proiectată pentru semnalul *X*, indicând o tranzacție cu o întârziere delta.

Atunci când timpul curent de simulare este între 5 ns și 10 ns, driverul pentru semnalul X are o singură tranzacție, ('0', T_c). La 10 ns, apare o tranziție a semnalului *B*, iar procesul este executat din nou. Prima instrucțiune secvențială nu determină adăugarea unei tranzacții la driver, astfel încât forma de undă proiectată a ieșirii nu este actualizată. La evaluarea condiției pentru instrucțiunea if, de data aceasta expresia este adevărată. Se execută asignarea $X \leq 11'$, ceea ce determină generarea unei noi tranzacții, ('1', 10 ns + δ), care se adaugă la driverul semnalului *X*.

Atunci când timpul de simulare ajunge la 15 ns, apare o tranziție a semnalului C, ceea ce determină execuția procesului. Prima instrucțiune determină adăugarea tranzacției ('0', 15 ns + δ) la driver. Însă, deoarece condiția instrucțiunii $\pm f$ este adevărată, următoarea asignare de semnal determină înlocuirea tranzacției curente cu ('1', 15 ns + δ). Deoarece valoarea '1' este aceeași cu valoarea curentă, nu este necesar ca această tranzacție să înlocuiască ultima tranzacție ('0', 15 ns + δ) în forma de undă proiectată a ieșirii, dacă ultima tranzacție este ștearsă.



Figura 2.5. Diagrama de timp pentru procesul proc2; a doua asignare determină înlocuirea unei tranzacții.

Atunci când timpul de simulare ajunge la 20 ns, semnalele B și C se modifică, iar procesul se execută din nou. Prima instrucțiune determină adăugarea tranzacției ('0', 20 ns + δ) la forma de undă proiectată a ieșirii. Condiția instrucțiunii *if* se evaluează la fals, iar procesul este suspendat.

O interpretare simplă a asignărilor la semnale din cadrul unui proces, presupunând că procesul nu conține clauze after, este următoarea:

- Toate expresiile se evaluează pe baza valorilor *curente* ale semnalelor din partea dreaptă a simbolului <=;
- Un semnal va fi actualizat cu valoarea din ultima asignare;
- Toate semnalele vor fi actualizate numai la sfârșitul procesului, atunci când acesta este suspendat.

Spre deosebire de instrucțiunile de asignare secvențiale din cadrul proceselor, o instrucțiune de asignare concurentă are o listă de sensibilitate implicită care cuprinde toate semnalele din partea dreaptă a simbolului <=. Instrucțiunile concurente se execută ori de câte ori are loc o tranziție a unui semnal din lista de sensibilitate implicită. Atunci când există instrucțiuni concurente multiple, ele nu se execută secvențial. În secțiunea următoare se va prezenta modul în care se execută instrucțiunile concurente multiple atunci când expresia care se evaluează conține semnale actualizate de o altă instrucțiune.

2.4.3 Ciclul de simulare

La simularea unui model VHDL, se execută mai întâi o fază de inițializare și apoi, în mod repetat, cicluri de simulare. Faza de inițializare începe cu timpul curent de simulare setat la 0 ns. În general, dacă pentru un semnal nu se specifică o valoare inițială în mod explicit, semnalul se va inițializa cu valoarea '0' dacă este de tip **BIT**, sau cu valoarea 'U' dacă este de tip **STD_LOGIC**. Apoi, se execută fiecare proces până când acesta va fi suspendat. Instrucțiunile concurente vor fi și ele executate.

După faza de inițializare, se execută cicluri de simulare, fiecare ciclu constând din următoarele etape:

- Se actualizează toate semnalele. Actualizarea va determina tranziții ale semnalelor și apariția unor evenimente pentru aceste semnale.
- Se execută fiecare din procesele ale căror liste de sensibilitate conțin semnale pentru care a apărut un eveniment în ciclul curent de simulare.
- Se determină timpul de simulare pentru următorul ciclu, T_u . Acest timp este fie momentul următor de timp la care un semnal își modifică valoarea, pe baza formei sale de undă proiectată a ieșirii, fie momentul de timp la care execuția unui proces este reluată (pentru modelele destinate simulării), în funcție de momentul care este mai apropiat. Dacă timpul de simulare pentru următorul ciclu este cu o întârziere delta sau cu un multiplu al întârzierii delta după timpul curent de simulare, atunci timpul curent de simulare, T_c , rămâne același și se execută un *ciclu delta* constând din aceleași etape ca cele de sus. Deci, o întârziere delta este de fapt o întârziere zero, doar în mod conceptual fiind convenabil să se considere că este o întârziere foarte mică. În caz contrar, timpul curent de simulare este setat la momentul de timp al următorului ciclu de simulare ($T_c = T_u$).

Pentru ilustrarea ciclului de simulare, considerăm descrierea din Exemplul 2.5. Modul de implementare al ciclului de simulare este specific diferitelor simulatoare, dar rezultatele simulării (de exemplu, formele de undă ale semnalelor) vor fi echivalente.

Exemplul 2.5

```
entity delta is
                                         -- 1
   port (A, B, C, D : in BIT;
                                        -- 2
        W, X, Y, Z : buffer BIT);
                                        -- 3
                                        ___
end delta;
                                            4
architecture delta of delta is
                                        ___
                                            5
begin
                                        ___
                                            6
   Z \leq not Y;
                                            7
                                        -- 8
   Y <= W or X;
   X <= C or D;
                                        ___
                                            9
   W \leq A and B;
                                        -- 10
                                        -- 11
end delta;
```

În faza de inițializare, toate semnalele sunt setate la '0'. Apoi, se execută fiecare instrucțiune concurentă. Ordinea scrierii și execuției instrucțiunilor concurente nu este importantă, ceea ce se va ilustra prin execuția instrucțiunilor începând cu ultima și terminând cu prima. Driverele semnalelor sunt actualizate conform formelor de undă proiectate din figura 2.6, fiind necesar un ciclu delta pentru actualizarea semnalelor înainte ca timpul curent de simulare să fie avansat.



Figura 2.6. Formele de undă proiectate ale semnalelor de intrare/ieșire pentru Exemplul 2.5.

Presupunem că tranzitia semnalelor de intrare este cea din figura 2.7. Atunci când timpul curent de simulare ajunge la 100 ns, începe un ciclu de simulare si semnalele sunt actualizate. Semnalul A trece din '0' în '1', ceea ce determină execuția instrucțiunii de asignare a semnalului W (linia 10). Valoarea semnalului W nu se modifică, astfel încât ciclul de simulare se termină, iar timpul de simulare poate avansa. Atunci când timpul curent de simulare ajunge la 200 ns, începe un nou ciclu de simulare. Semnalul B trece din '0' în '1' și se va executa instrucțiunea de asignare din linia 10. Se adaugă o nouă tranzacție ('1', 200 ns + δ) la driverul semnalului W. Este necesar un ciclu delta, astfel încât timpul de simulare nu avansează. În timpul acestui ciclu delta, semnalul W este actualizat cu noua sa valoare. Aceasta determină execuția instrucțiunii din linia 8. Se adaugă o nouă tranzacție la driverul semnalului Y, ('1', 200 ns + δ). Este necesar un al doilea ciclu delta, în care semnalul Y este actualizat cu noua sa valoare. Aceasta determină execuția instrucțiunii din linia 7. Se adaugă o nouă tranzacție la driverul semnalului Z, ('0', 200 ns + δ). Este necesar un al treilea ciclu delta pentru actualizarea semnalului Z, după care timpul curent de simulare poate avansa.



Figura 2.7. Formele de undă ale semnalelor de intrare și de ieșire pentru Exemplul 2.5.

Tabelul 2.2 prezintă ciclurile de simulare și ciclurile delta necesare pentru simularea descrierii din Exemplul 2.5 pentru intervalul de timp între 0 și 400 ns.

Ciclu de simulare	ns	δ	Α	В	С	D	W	X	Y	Ζ
1	000	+0	0	0	0	0	0	0	0	0
	000	+1	0	0	0	0	0	0	0	1
2	100	+0	1	0	0	0	0	0	0	1
3	200	+0	1	1	0	0	0	0	0	1
	200	+1	1	1	0	0	1	0	0	1
	200	+2	1	1	0	0	1	0	1	1
	200	+3	1	1	0	0	1	0	1	0
4	300	+0	0	1	0	0	1	0	1	0
	300	+1	0	1	0	0	0	0	1	0
	300	+2	0	1	0	0	0	0	0	0
	300	+3	0	1	0	0	0	0	0	1
5	400	+0	0	0	0	0	0	0	0	1

Tabelul 2.2. Cicluri de simulare și cicluri delta pentru Exemplul 2.5.

2.4.4 Drivere multiple și funcții de rezoluție

Limbajul VHDL nu permite ca un semnal să aibă mai mult de un driver fără să existe o funcție de rezoluție asociată semnalului. O funcție de rezoluție este utilizată pentru a determina o valoare a unui semnal pe baza driverelor multiple. Figura 2.8 ilustrează un caz în care un semnal are două drivere.



Figura 2.8. Semnal cu drivere multiple.

În funcție de ieșirile curente ale celor două porți, valoarea logică a semnalului *Y* poate fi '0', '1' sau 'X', unde 'X' este o valoare nedeterminată. 'X' nu este o valoare indiferentă, în modul în care aceasta este utilizată adesea pentru minimizarea logică, ci este o valoare necunoscută. Circuitul din figura 2.8 poate fi descris în limbajul VHDL în modul indicat în Exemplul 2.6.

Exemplul 2.6

```
architecture driver_multiplu of exemplu is
begin
    Y <= A and B;
    Y <= A or B;
end driver multiplu;
```

În acest exemplu, cele două instrucțiuni concurente de asignare determină crearea a două drivere pentru semnalul Y. Deși două asignări secvențiale la același semnal în același proces nu creează drivere separate, ci numai tranzacții pentru același driver, asignările secvențiale la același semnal în procese diferite au același efect ca și cele două instrucțiuni concurente de asignare din exemplul anterior. Semnalului Y trebuie să i se asocieze o funcție de rezoluție. Această funcție va fi utilizată pentru determinarea valorii semnalului Y pe baza valorilor driverelor. Conceptul de rezoluție a două drivere este exemplificat în tabelul 2.3. Dacă cele două drivere sunt aceleași, semnalul de ieșire este cunoscut; în caz contrar, acest semnal are o valoare necunoscută.

Α	В	A and B	A or B	Y
0	0	0	0	0
0	1	0	1	'X'
1	0	0	1	'X'
1	1	1	1	1

Tabelul 2.3. Tabel de adevăr pentru rezoluția unui semnal cu drivere multiple.

Este mai puțin probabil că se va scrie în mod intenționat un cod ca cel din Exemplul 2.6 în care driverele sunt în conflict. Aplicațiile cele mai obișnuite pentru care se utilizează funcții de rezoluție sunt cele cu magistrale. De exemplu, în cazul unei magistrale de date pot exista mai multe entități care pot reprezenta drivere pentru această magistrală. Pentru evitarea conflictelor, nu trebuie să existe simultan două drivere pentru magistrală. Totuși, dacă există două entități conectate la magistrală care generează simultan nivele logice opuse, este util ca simulatoarele să poată indica un conflict prin calcularea unei valori corespunzătoare, cum este valoarea 'X'. O funcție de rezoluție pentru semnalul Y din Exemplul 2.6 poate fi modelată prin fragmentul următor:

```
function CALCUL_VAL (A, B : X01) return X01 is
begin
    if A /= B then return ('X');
        else return (A);
    end if;
end CALCUL VAL;
```

Dacă Y este un semnal asupra căruia se aplică această funcție de rezoluție, în cazul în care două drivere sunt în conflict, semnalului Y i se va asigna valoarea 'X'.

2.5 Simularea utilizând bancuri de test

Pentru sisteme digitale de complexitate redusă și medie, cea mai utilizată metodă pentru testarea funcțională a circuitelor componente și a întregului sistem este simularea cu ajutorul bancurilor de test. Un asemenea banc de test va aplica o secvență de valori la intrările circuitului testat și va afișa ieșirile generate de circuit sub formă alfanumerică sau grafică. Mai precis, bancul de test va aplica o secvență de vectori de test la intrările proiectului testat (DUT – *Design Under Test*) și va afișa semnalele de ieșire care sunt generate în urma simulării. Opțional, bancul de test poate compara în mod automat valorile semnalelor de ieșire rezultate în urma simulării cu valorile așteptate (corecte) ale acestor semnale.

2.5.1 Funcțiile și structura unui banc de test

Simulatoarele limbajelor de descriere hardware permit, în general, utilizarea unei metode grafice pentru asignarea valorii vectorilor de test. Totuși, utilizarea unei metode grafice are dezavantajul că este necesar un timp relativ lung pentru a crea secvențe de test mai lungi. De asemenea, aceste secvențe de test nu pot fi repetate întotdeauna după modificarea circuitului testat și re-executarea simulării. În plus, un banc de test creat astfel pentru un anumit simulator nu va putea fi utilizat dacă în viitor se va utiliza un alt simulator pentru testare.

Metoda recomandată pentru simulare prin utilizarea bancurilor de test este de a se crea bancurile de test ca module scrise în limbajul de descriere hardware utilizat pentru descrierea circuitelor testate. Prin aceasta, bancurile de test vor fi portabile între diferite medii de proiectare și vor putea fi reutilizate în proiectele viitoare. Un banc de test va avea proiectul testat (DUT) ca un sub-modul, iar toate semnalele de intrare ale acestui proiect vor fi generate în interiorul bancului de test. Dacă se utilizează verificarea automată, atunci valorile semnalelor de ieșire ale proiectului testat

vor fi comparate cu valorile corecte ale acestor semnale. De exemplu, aceste valori corecte pot fi calculate în interiorul bancului de test.

Funcțiile principale ale unui banc de test sunt următoarele:

- Instanțierea proiectului testat;
- Generarea și aplicarea vectorilor de test la intrările proiectului testat;
- Afișarea valorii semnalelor de ieșire rezultate în urma simulării;
- Opțional, compararea valorii semnalelor de ieșire cu cele așteptate.

Structura generală a unui banc de test este ilustrată în figura 2.9. Bancul de test conține un modul pentru generarea vectorilor de test, proiectul testat DUT, un modul pentru afișarea rezultatelor și un modul pentru verificarea rezultatelor.



Figura 2.9. Structura generală a unui banc de test.

Modulul VHDL al unui banc de test este unul special, deoarece entitatea acestuia nu are porturi de intrare și de ieșire. Deoarece un asemenea modul se utilizează numai pentru simulare și nu se va sintetiza într-un circuit, acesta nu este limitat la subsetul limbajului care se poate utiliza doar pentru sinteză. Astfel, se poate utiliza orice construcție funcțională a limbajului pentru a crea configurații complexe ale vectorilor de test și pentru a efectua calculele necesare verificării rezultatelor. Exemple de instrucțiuni utile pentru scrierea bancurilor de test sunt wait, assert (cu clauza report) și instrucțiunea report separată (începând cu versiunea VHDL-93 a limbajului).

În general, un banc de test VHDL conține următoarele secțiuni principale:

- Declarația entității și a arhitecturii;
- Declarațiile semnalelor reprezentând intrările și ieșirile proiectului testat;
- Instanțierea entității proiectului testat;
- Pentru circuite secvențiale, un proces pentru generarea semnalului de ceas;
- Un proces pentru generarea vectorilor de test;
- Un proces pentru afișarea și verificarea rezultatelor.

Un exemplu de banc de test este prezentat în secțiunea 2.5.6.

2.5.2 Generarea semnalului de ceas

Pentru circuite sau sisteme secvențiale, este necesară generarea semnalului de ceas care se va aplica la intrările circuitului sau sistemului testat. Pentru generarea semnalului de ceas se pot utiliza mai multe metode, două dintre acestea fiind prezentate în continuare. Prima metodă utilizează clauza after, iar a doua metodă utilizează instrucțiunea wait.

```
-- Declararea unei constante cu perioada semnalului de ceas
constant CLK_PERIOD : TIME := 10 ns;
-- Metoda 1
Clk <= not Clk after CLK_PERIOD/2;
-- Metoda 2
gen_clk: process
begin
Clk <= '0';
wait for (CLK_PERIOD/2);
Clk <= '1';
wait for (CLK_PERIOD/2);
end process gen_clk;
```

2.5.3 Generarea vectorilor de test

Pentru testarea completă a funcționării unui circuit, ar trebui să se aplice circuitului toate combinațiile posibile ale intrărilor. Pentru multe circuite, o asemenea testare completă nu este însă posibilă. De exemplu, numărul combinațiilor posibile ale intrărilor poate fi prea mare, sau anumite combinații ale intrărilor nu sunt permise. De aceea, generarea secvenței vectorilor de test este dependentă de circuitul testat.

Atunci când numărul combinațiilor intrărilor de test este prea mare, există mai multe posibilități pentru generarea vectorilor de test. În primul rând, pe baza cunoașterii funcțiilor circuitului se pot elimina combinațiile de intrare care nu sunt relevante sau care nu vor apare în practică. În al doilea rând, circuitul poate fi partiționat în mai multe sub-circuite, fiecare fiind testat exhaustiv. Apoi, se poate testa întregul circuit utilizând un set de test care nu este exhaustiv pentru a verifica faptul că sub-circuitele au fost integrate corect.

În al treilea rând, o posibilitate de generare a vectorilor de test este alegerea unui set reprezentativ de cazuri de test care să acopere condițiile extreme și condițiile considerate normale (diferite de cele extreme). Condițiile extreme reprezintă, de exemplu, acele cazuri în care toate intrările de test au valorile minime posibile sau toate intrările de test au valorile maxime posibile. Generarea acestor cazuri de test pentru condițiile extreme și cele normale ar trebui realizată într-un mod sistematic. De exemplu, se pot genera vectori de test cu intrările având valorile minime și câteva valori mai mari decât acestea, cu intrările având valorile maxime și câteva valori decât acestea, și cu intrările având un set de valori oarecare între valorile minime și maxime.

2.5.4 Afişarea rezultatelor simulării

Pentru afișarea rezultatelor simulării se pot utiliza instrucțiunile assert și report. Instrucțiunea assert (prezentată în secțiunea 2.3.3) testează o condiție și, în cazul în care condiția testată este falsă, afișează șirul de caractere specificat după clauza report a acestei instrucțiuni. Pentru afișarea necondiționată a unui șir de caractere în timpul simulării, începând cu versiunea VHDL-93 a limbajului clauza report se poate utiliza ca o instrucțiune completă, fără a fi necesară testarea unei condiții.

Pentru afișarea unor valori cu instrucțiunile assert și report, valorile respective trebuie convertite în șiruri de caractere. Pentru conversie se poate utiliza atributul 'image. Acest atribut este definit începând cu versiunea VHDL-93 a limbajului și permite conversia unei valori scalare (BIT, INTEGER, STD_LOGIC) în șirul de caractere corespunzător valorii, șir care se poate utiliza ca argument al instrucțiunilor assert sau report. Pentru valori vectoriale (BIT_VECTOR, SIGNED, UNSIGNED, STD_LOGIC_VECTOR) este necesară utilizarea unor funcții de conversie. Atributul 'image se poate utiliza sub forma *tip*'image (*expresie*), unde *expresie* trebuie să fie de un tip scalar corespunzător prefixului *tip*.

Pentru concatenarea mai multor șiruri de caractere, se poate utiliza operatorul de concatenare &. Șirul rezultat prin concatenare se poate utiliza ca un singur argument al unei instrucțiuni assert sau report, în modul ilustrat în exemplul din secțiunea 2.5.6.

Pentru afișarea timpului de simulare, se poate utiliza funcția predefinită now. Această funcție returnează timpul curent de simulare, care este de tip TIME. De exemplu, expresia TIME'image (now) este reprezentarea sub forma unui șir de caractere a timpului curent de simulare.

2.5.5 Verificarea automată a rezultatelor simulării

Atunci când la testarea unui circuit se utilizează un număr mare de vectori de test, este recomandată verificarea automată a răspunsului corect al circuitului la fiecare vector de test. Se pot utiliza mai multe metode pentru această verificare automată. Una din metode constă în compararea rezultatelor simulării cu răspunsurile corecte ale circuitului. Pentru aceasta, mai întâi se creează un fișier de referință conținând secvența răspunsurilor corecte ale circuitului. Apoi, rezultatele simulării sunt înscrise într-un fișier și se compară conținutul acestui fișier cu cel al fișierului de referință. Dezavantajul acestei metode constă în dificultatea identificării sursei erorii pe baza răspunsului eronat al circuitului.

O altă metodă de verificare automată constă în compararea fiecărui răspuns al circuitului cu răspunsul corect în timpul simulării, și nu la sfârșitul întregului proces de simulare. În bancul de test se pot include instrucțiuni pentru identificarea momentului de timp sau a vectorului de intrare pentru care răspunsul circuitului este eronat, ceea ce permite identificarea sursei erorii. Răspunsurile corecte pot fi memorate într-un fișier sau pot fi calculate în timpul simulării. Atunci când răspunsurile corecte sunt calculate în timpul simulării, pentru calcul trebuie să se utilizeze o metodă complet diferită de cea utilizată de circuitul testat, deoarece în caz contrar se va repeta aceeași greșeală în ambele calcule.

Compararea răspunsurilor reale cu cele corecte trebuie să se realizeze la anumite intervale de timp. În cazul circuitelor secvențiale, compararea se poate realiza, de exemplu, la fiecare front crescător al semnalului de ceas sau la un interval de un anumit număr de cicluri de ceas. În cazul circuitelor combinaționale, la stabilirea intervalului de timp trebuie să se țină cont de întârzierile combinaționale ale circuitului respectiv.

2.5.6 Exemplu de banc de test

Se prezintă ca exemplu un banc de test pentru un modul combinațional al unui sumator elementar. Descrierea sumatorului elementar prin ecuațiile acestuia este prezentată în Exemplul 2.7.

Exemplul 2.7

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity SE is
    port (X : in STD_LOGIC;
        Y : in STD_LOGIC;
        Tin : in STD_LOGIC;
        S : out STD_LOGIC;
        Tout : out STD_LOGIC;
        Tout : out STD_LOGIC);
end SE;
architecture sumator of SE is
begin
    S <= X xor Y xor Tin;
    Tout <= (X and Y) or (X and Tin) or (Y and Tin);
end sumator;
```

Exemplul 2.8 conține un modul al unui banc de test pentru sumatorul elementar. În acest exemplu, se utilizează un singur proces pentru generarea vectorilor de test și verificarea rezultatelor. În general, se pot utiliza procese separate pentru aceste două secțiuni ale bancului de test, sau se poate separa secțiunea de verificare a rezultatelor într-o procedură sau funcție care este apelată din procesul care generează vectorii de test.

Exemplul 2.8

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

```
entity SE tb is
                                            -- bancul de test nu are
end SE tb;
                                            -- porturi de intrare si iesire
architecture SE tb of SE tb is
-- Declaratiile semnalelor de intrare
signal X : STD LOGIC := '0';
signal Y : STD LOGIC := '0';
signal Tin : STD LOGIC := '0';
-- Declaratiile semnalelor de iesire
signal S : STD LOGIC;
signal Tout : STD LOGIC;
begin
  -- Instantierea entitatii proiectului testat (DUT)
  DUT: entity WORK.SE port map (
          X => X,
           Y = Y,
          Tin => Tin,
           S => S,
           Tout => Tout);
   -- Generarea vectorilor de test si verificarea rezultatelor
  gen vect test: process
  variable VecTest : STD LOGIC VECTOR (2 downto 0); -- vector de test
  variable RezCorect : STD LOGIC VECTOR (1 downto 0); -- rezultat asteptat
  variable NrErori : INTEGER := 0;
                                                       -- numar de erori
  begin
     VecTest := "000";
      for i in 0 to 7 loop
                                                       -- genereaza toate
                                                       -- combinatiile
        X <= VecTest(2);
        Y <= VecTest(1);
                                                       -- pentru X, Y, Tin
        Tin <= VecTest(0);
        wait for 10 ns;
        RezCorect := ('0' \& X) + ('0' \& Y) + ('0' \& Tin);
        if ((Tout & S) /= RezCorect) then -- verifica
            report "Rezultat asteptat (" &
                                                       -- rezultatele
              STD LOGIC'image (RezCorect(1)) &
              STD LOGIC'image (RezCorect(0)) &
              ") /= Valoare obtinuta (" &
               STD LOGIC'image (Tout) & STD LOGIC'image (S) &
              ") la t = " & TIME'image (now)
               severity ERROR;
           NrErori := NrErori + 1;
        end if;
        VecTest := VecTest + 1;
     end loop;
      report "Testare terminata cu " &
        INTEGER'image (NrErori) & " erori";
     wait;
   end process gen_vect_test;
end;
```

2.6 Exemplu de utilizare a simulatorului Vivado

2.6.1 Prezentare generală a simulatorului

Simulatorul Vivado este un simulator bazat pe evenimente care permite simularea funcțională și temporală a proiectelor descrise în limbajul VHDL, Verilog sau SystemVerilog. Simulatorul este integrat în mediul de proiectare Vivado Design Suite și permite executarea următoarelor operații:

- Analiza fișierelor sursă și stocarea fișierelor analizate într-o bibliotecă HDL. Dacă limbajul de descriere utilizat este VHDL, analiza se poate realiza prin comanda xvhdl.
- Elaborarea și linkeditarea descrierii HDL utilizând comanda xelab. Pentru un anumit modul de nivel superior, această comandă încarcă unitățile de proiectare ale tuturor sub-modulelor, compilează fișierele sursă, generează un cod executabil și linkeditează codul generat cu nucleul de simulare pentru a crea un instantaneu de simulare executabil.
- Încărcarea unui instantaneu de simulare și executarea simulării. Comanda pentru executarea acestei operații este xsim.

Simulatorul Vivado permite compilarea fișierelor sursă utilizând fire multiple de execuție, depanarea la nivelul codului sursă (execuția pas cu pas, setarea unor puncte de întrerupere, afișarea valorilor curente ale semnalelor și variabilelor) și vizualizarea rezultatelor simulării sub formă grafică.

2.6.2 Descrierea exemplului de simulare

În secțiunile următoare se prezintă etapele care trebuie executate pentru simularea funcționării unui numărător binar de 4 biți cu o intrare de validare, care numără în sus (de la 0000 la 1111).

Descrierea în limbajul VHDL a numărătorului este prezentată în Exemplul 2.9. Pentru actualizarea valorii numărătorului se utilizează o funcție de incrementare a unui vector de biți.

Exemplul 2.9

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity num4 is
    port (Clk : in STD_LOGIC;
        Rst : in STD_LOGIC;
        En : in STD_LOGIC;
        Num : out STD_LOGIC_VECTOR (3 downto 0));
end num4;
```

```
architecture simul of num4 is
function INC BV (A : STD LOGIC VECTOR) return STD LOGIC VECTOR is
  variable Rez : STD LOGIC VECTOR (A'range);
  variable C : STD LOGIC;
begin
   C := '1';
  for i in A'range loop
     Rez(i) := A(i) xor C;
     C := A(i) and C;
   end loop;
  return Rez;
end INC BV;
signal Num int : STD LOGIC VECTOR (3 downto 0);
begin
  process (Clk)
  begin
     if (Clk'event and Clk = '1') then
        if (Rst = '1') then
           Num int <= (others => '0');
         elsif(En = '1') then
          Num int <= INC BV (Num int);
        end if;
     end if;
   end process;
  Num <= Num int;
end simul;
```

2.6.3 Crearea proiectului și a fișierului sursă

Executați operațiile descrise în continuare pentru crearea unui nou proiect și a fișierului sursă.

Observație

Etapele descrise în continuare presupun utilizarea versiunii 2010.1 a mediului de proiectare Xilinx Vivado. Pot fi necesare modificări ale acestor etape dacă se utilizează o versiune diferită.

- 1. Lansați în execuție aplicația Vivado.
- 2. În fereastra aplicației Vivado, secțiunea *Quick Start*, selectați **Create Project**. Se afișează fereastra de dialog *Create a New Vivado Project*.
- 3. Selectați butonul Next în fereastra *Create a New Vivado Project*. Se afișează fereastra de dialog *Project Name*.
- 4. În câmpul *Project name*, introduceți numele proiectului (de exemplu, **num4**). În câmpul *Project location*, selectați butonul ..., navigați la directorul în care trebuie creat proiectul, iar apoi selectați butonul **Select**. Verificați ca opțiunea

Create project subdirectory să fie bifată și selectați butonul **Next**. Se afișează fereastra de dialog *Project Type*.

- Verificați ca tipul proiectului selectat să fie RTL Project, verificați ca opțiunea Do not specify sources at this time să nu fie bifată, iar apoi selectați butonul Next. Se afişează fereastra de dialog *Add Sources*.
- 6. Verificați ca pentru opțiunile *Target language* și *Simulator language* să fie selectat limbajul **VHDL**, după care selectați butonul **Create File**. Se va afișa caseta de dialog *Create Source File*.
- 7. În câmpul *File type* verificați să fie selectat limbajul **VHDL**, introduceți numele fișierului (de exemplu, **num4**) în câmpul *File name* și selectați butonul **OK**.
- 8. În aceeași fereastră de dialog Add Sources, selectați butonul Next.
- 9. În fereastra de dialog Add Constraints (optional), selectați butonul Next.
- 10. În fereastra de dialog *Default Part*, selectați panoul **Boards**, în tabelul afișat selectați placa dorită (de exemplu, **Nexys4 DDR** sau **Nexys A7-100T**), iar apoi selectați butonul **Next**.
- 11. În fereastra *New Project Summary*, revizuiți setările cu care se va crea noul proiect și selectați butonul **Finish** pentru crearea proiectului.
- 12. În fereastra de dialog *Define Module*, selectați butonul **OK**, fără a introduce porturile de I/E. Confirmați această opțiune selectând butonul **Yes** în caseta de dialog *Define Module*.
- 13. Deschideți fișierul sursă creat în fereastra de editare executând un clic dublu pe numele acestui fișier în panoul *Hierarchy* din fereastra *Sources*.
- 14. Înlocuiți conținutul ferestrei de editare pentru fișierul sursă creat anterior cu descrierea în limbajul VHDL a numărătorului din Exemplul 2.9. Salvați fișierul sursă selectând iconița *Save File* din colțul din stânga sus al ferestrei de editare.

2.6.4 Crearea modulului pentru bancul de test

Pentru crearea unui modul al bancului de test, executați operațiile descrise în continuare.

1. Selectați opțiunea **Add Sources** din panoul *Flow Navigator*. În fereastra de dialog *Add Sources*, selectați opțiunea **Add or create simulation sources** și selectați butonul **Next**. Se afișează fereastra de dialog *Add or Create Simulation Sources*.

- 2. Selectați butonul Create File; se va afișa caseta de dialog Create Source File. Verificați să fie selectată opțiunea VHDL în câmpul File type și introduceți numele fișierului în câmpul File name. Acest nume trebuie să fie diferit de numele fișierului sursă VHDL. De exemplu, dacă numele fișierului sursă este num4, numele fișierului pentru bancul de test poate fi num4_tb. Selectați butonul OK.
- 3. În fereastra de dialog *Add or Create Simulation Sources* selectați butonul **Finish**. Se va afișa fereastra de dialog *Define Module*.
- 4. Deoarece un modul al bancului de test nu are porturi de intrare şi de ieşire, în fereastra de dialog *Define Module* nu este necesară introducerea acestor porturi. Selectați butonul **OK** şi confirmați această opțiune selectând butonul **Yes** în caseta de dialog *Define Module*.
- 5. În panoul *Hierarchy* din fereastra *Sources*, expandați intrarea *Simulation Sources*, expandați intrarea *sim_1* și deschideți fișierul bancului de test creat anterior (de exemplu, num4_tb) în fereastra de editare.
- 6. În fișierul bancului de test, adăugați o nouă clauză use după cea existentă, specificând utilizarea pachetului **STD_LOGIC_UNSIGNED** din biblioteca IEEE.
- 7. În partea declarativă a arhitecturii (înainte de cuvântul cheie begin), declarați semnalele de intrare pentru proiectul testat, având același nume și același tip cu cel al porturilor de intrare ale numărătorului, și inițializați fiecare semnal cu '0'.
- 8. Declarați semnalul de ieșire al proiectului testat, având același nume și același tip cu cel al portului de ieșire al numărătorului.
- 9. După declarația semnalelor, declarați o constantă pentru perioada semnalului de ceas de 10 ns, în modul în care este definită constanta CLK_PERIOD în secțiunea 2.5.2.
- 10. În corpul arhitecturii (după cuvântul cheie begin), instanțiați entitatea numărătorului, conectând la porturile sale semnalele de intrare și de ieșire declarate anterior. Utilizați eticheta DUT, care va reprezenta numele instanței numărătorului.
- 11. După instanțierea numărătorului, scrieți un proces pentru generarea semnalului de ceas, utilizând metoda 2 prezentată în secțiunea 2.5.2.
- 12. În continuare, scrieți un proces (fără o listă de sensibilitate) pentru generarea semnalelor de intrare necesare funcționării numărătorului și pentru verificarea ieșirilor acestuia. În partea declarativă a acestui proces, declarați mai întâi o variabilă pentru ieșirea corectă (cea așteptată) a numărătorului și o variabilă pentru numărul de erori obținute la simulare, în modul în care sunt definite variabilele *RezCorect* și *NrErori* din Exemplul 2.8, cu deosebirea că variabila pentru ieșirea corectă RezCorect trebuie să fie de 4 biți și trebuie inițializată cu "0000".

13. În corpul procesului, introduceți următoarele instrucțiuni pentru generarea unui impuls de resetare a numărătorului și pentru validarea funcționării acestuia:

```
Rst <= '1';
wait for CLK_PERIOD;
Rst <= '0';
wait for CLK_PERIOD;
En <= '1';</pre>
```

- 14. În continuarea procesului, scrieți o buclă for loop pentru verificarea fiecăreia din cele 16 ieșiri posibile ale numărătorului. În această buclă, dacă ieșirea numărătorului este diferită de ieșirea corectă, afișați ieșirea corectă, ieșirea obținută și timpul de simulare la care a apărut diferența (cu o instrucțiune report), după care incrementați variabila reprezentând numărul de erori. Înainte de sfârșitul buclei, adăugați o instrucțiune wait for pentru a aștepta un timp corespunzător perioadei semnalului de ceas și incrementați variabila reprezentând ieșirea corectă a numărătorului.
- 15. După bucla for loop, verificați dacă au apărut erori la simulare. Dacă nu au fost erori, afișați un mesaj indicând faptul că simularea s-a terminat cu succes. În caz contrar, afișați un mesaj indicând faptul că simularea s-a terminat cu erori și afișați numărul de erori apărute.
- 16. Încheiați procesul printr-o instrucțiune wait pentru suspendarea procesului.
- 17. Salvați fișierul bancului de test selectând iconița *Save File* . Verificați să nu fie erori de sintaxă afișate în fereastra *Sources*.

2.6.5 Setarea opțiunilor pentru simulare

Pentru modificarea opțiunilor de simulare, executați operațiile următoare.

- 1. În panoul *Flow Navigator*, selectați opțiunea **Settings**. Se va afișa panoul *General* din zona *Project Settings* a ferestrei de dialog *Settings*.
- 2. În fereastra de dialog *Settings*, selectați linia **Simulation** din zona *Project Settings*.
- 3. Vizualizați opțiunile disponibile pentru proprietatea *Target simulator*, dar nu modificați simulatorul care este selectat în mod implicit, *Vivado Simulator*.
- 4. Verificați ca pentru proprietatea *Simulator language* să fie selectată opțiunea **VHDL**.
- 5. În partea dreaptă a ferestrei de dialog *Settings*, selectați panoul *Simulation*. Modificați proprietatea xsim.simulate.runtime, care indică timpul pentru care se va executa simularea, de la 1000ns la **200ns**.
- 6. Selectați butonul OK pentru a închide fereastra de dialog Project Settings.

2.6.6 Executarea simulării

Pentru executarea simulării funcționale a numărătorului, parcurgeți etapele descrise în continuare.

- În panoul *Flow Navigator*, secțiunea SIMULATION, selectați comanda Run Simulation și selectați singura opțiune disponibilă, Run Behavioral Simulation. Celelalte tipuri de simulare vor fi disponibile după execuția etapelor de sinteză, respectiv de implementare. Mediul de proiectare Vivado va lansa în execuție comanda xvhdl pentru analiza proiectului, după care va lansa în execuție comanda xelab pentru elaborarea, compilarea și linkeditarea proiectului întrun instantaneu de simulare executabil. La terminarea acestor operații, mediul de proiectare Vivado va lansa în execuție comanda xsim pentru execuția simulării. Dacă nu sunt erori, rezultatele simulării vor fi afișate sub formă grafică într-o fereastră *Wave*, în partea dreaptă a ecranului.
- 2. Selectați iconița *Zoom Fit* 🔀 din meniul ferestrei *Wave* pentru afișarea formei de undă a semnalelor pentru întregul interval de simulare.
- 3. În fereastra *Wave* se poate observa că secvența de numărare obținută, indicată prin succesiunea valorilor hexazecimale ale semnalului Num, este 0, 8, 4, c, 2, a, ..., secvență care nu este corectă (figura 2.10).



Figura 2.10. Rezultatele simulării afișate în fereastra Wave a simulatorului Vivado.

- 4. Adăugați semnalul intern Num_int în fereastra Wave. Pentru aceasta, în panoul Scope al domeniilor de valabilitate selectați entitatea testată DUT (expandați entitatea bancului de test dacă este necesar). Porturile şi semnalele interne ale entității testate vor apare în panoul Objects. În acest panou, executați un clic cu butonul din dreapta pe numele semnalului Num_int[3:0] şi selectați opțiunea Add to Wave Window. Se poate observa că pentru noul semnal adăugat forma de undă nu a fost afişată. Pentru afişarea formei de undă a tuturor semnalelor adăugate în fereastra Wave simularea trebuie repornită şi executată din nou.
- 5. Executați un clic cu butonul din dreapta pe numele semnalului Num_int[3:0] din fereastra *Wave*, selectați opțiunea **Radix**, iar apoi selectați **Binary**.

- 6. În meniul principal Vivado, selectați comanda *Run* → *Restart* pentru repornirea simulării.
- În panoul *Tcl Console* din partea de jos a ecranului, introduceți comanda run 200 ns în linia de comandă și apăsați tasta *Enter*. Forma de undă a semnalului Num_int va fi acum vizibilă în fereastra *Wave*.
- 8. Examinați mesajele afișate în panoul *Tcl Console*. Se observă afișarea unor mesaje de eroare indicând faptul că ieșirea corectă a numărătorului este diferită de ieșirea obținută la simulare. Se poate observa că valoarea binară a unei anumite ieșiri obținute corespunde valorii binare a ieșirii corecte corespunzătoare în cazul în care biții ieșirii obținute sunt citiți în ordine inversă, începând cu bitul cel mai puțin semnificativ. Același lucru se poate observa în fereastra *Wave*, examinând valorile binare afișate ale semnalului Num_int.
- 9. Închideți sesiunea de simulare selectând opțiunea File → Close Simulation din meniul principal şi confirmați închiderea în caseta de dialog Confirm Close. În caseta de dialog Save Waveform Configuration selectați butonul Discard pentru a nu salva configurația ferestrei Wave.

2.6.7 Corectarea erorii

Pentru corectarea descrierii numărătorului, deschideți pentru editare fișierul sursă al acestuia. Se poate observa că în funcția INC_BV vectorul de biți transmis ca parametru este prelucrat în ordinea corespunzătoare domeniului vectorului, cea indicată de atributul 'range. Pentru incrementarea corectă a vectorului, bucla for loop trebuie parcursă începând cu bitul cel mai puțin semnificativ al vectorului. Deci, eroarea se poate corecta prin modificarea liniei sursă

```
for i in A'range loop
```

astfel:

```
for i in A'low to A'high loop
```

Corectați fișierul sursă și salvați fișierul selectând iconița Save File 📕.

2.6.8 Re-executarea simulării

Pentru re-executarea simulării funcționale a numărătorului, executați următoarele operații:

- 1. În panoul *Flow Navigator*, secțiunea SIMULATION, selectați comanda **Run** Simulation și selectați opțiunea **Run Behavioral Simulation**.
- 2. Selectați iconița *Zoom Fit* 🔀 din meniul ferestrei *Wave* pentru afișarea formei de undă a semnalelor pentru întregul interval de simulare.

- 3. Se poate observa că secvența de numărare este acum cea corectă. În panoul *Tcl Console* nu ar mai trebui să se afișeze mesaje de eroare.
- 4. Închideți sesiunea de simulare selectând iconița *Close Simulation* ⊠ din colțul din dreapta sus al ferestrei *Behavioral Simulation* și confirmați închiderea în caseta de dialog *Confirm Close*.

Aplicații

- 2.1 Răspundeți la următoarele întrebări:
 - a. Care sunt avantajele simulării asistate de calculator?
 - b. Care sunt principalele tipuri de simulare?
 - c. Care sunt tipurile de intrări necesare pentru un simulator și ce metode pot fi utilizate pentru specificarea acestor intrări?
 - d. Care sunt metodele pentru verificarea automată a rezultatelor simulării într-un banc de test?
- **2.2** Explicați următoarele noțiuni: simulare bazată pe evenimente; întârziere delta; ciclu delta; formă de undă proiectată a ieșirii.
- **2.3** Executați etapele descrise în secțiunea 2.6 pentru simularea funcționării numărătorului binar de patru biți.
- 2.4 Creați un proiect pentru un modul VHDL care convertește o intrare binară de patru biți în două cifre zecimale reprezentate în cod BCD. Intrarea modulului este valoarea binară *Hex* de patru biți, iar ieșirile sunt cifrele zecimale *BCD1* și *BCD0* de câte patru biți, *BCD1* fiind cifra cea mai semnificativă. De exemplu, dacă intrarea *Hex* este x"5", ieșirile *BCD1* și *BCD0* sunt x"0", respectiv x"5", iar dacă intrarea *Hex* este x"C", ieșirile *BCD1* și *BCD0* sunt x"1", respectiv x"2". Creați apoi un modul al bancului de test pentru modulul de conversie, similar cu modulul bancului de test din Exemplul 2.8. Generați toate intrările posibile ale modulului și verificați dacă ieșirile generate sunt corecte. Simulați funcționarea modulului de conversie cu simulatorul Vivado.

Observație

Utilizați instrucțiuni concurente de asignare condițională (when .. else) pentru scrierea modulului de conversie. Instrucțiunile de asignare condițională sunt prezentate în Anexa B, *Instrucțiuni concurente în limbajul VHDL*, secțiunea B.1.3.2.

2.5 Creați un proiect pentru un numărător binar sincron de patru biți care numără în sus. Numărătorul are o intrare de validare En, o intrare de ceas Clk și o intrare de resetare sincronă Rst. Ieșirea numărătorului este vectorul Q de patru biți.

Pentru realizarea numărătorului, utilizați bistabile de tip T conectate ca în figura 2.11, cu n = 4. Fiecare bistabil T comută starea ieșirii Q la fiecare impuls de ceas, dacă intrarea de date T este activată. Creați mai întâi un modul VHDL pentru un bistabil de tip T. Creați apoi un modul principal pentru numărător, în care instanțiați entitatea bistabilului conform conexiunilor din figura 2.11. Creați un modul al bancului de test pentru numărător, similar cu modulul bancului de test creat în secțiunea 2.6.4. Simulați funcționarea numărătorului cu simulatorul Vivado.



Figura 2.11. Schema unui numărător binar sincron de *n* biți realizat cu bistabile T.