Lucrare de laborator nr. 9
Modelarea structurală ordonată şi modelarea comportamentală în VHDL

1. Scopul lucrării

Însuşirea principiilor pentru descrierea circuitelor cu structură ordonată de componente cu ajutorul instrucțiunii *generate*. Însuşirea principiilor şi tehnicilor privind modelarea comportamentală în VHDL a circuitelor digitale: instrucțiunea *process*, instrucțiuni secvențiale. Sunt exemplificate modelul structural al unui sumator pe 8 biți, respectiv modelul comportamental al unui registru de deplasare configurabil. Pentru fiecare model sunt prezentate arhitecturi de test adecvate pentru simularea şi verificarea modelelor respective.

Pentru buna desfăşurare a acestei lucrări de laborator este necesară citirea şi însuşirea noțiunilor din lucrările de laborator anterioare precum şi a notelor de curs, în special cele referitoare la instrucțiunea *generate*, instrucțiunea *process* şi instrucțiunile secvențiale.

2. Modelarea structurală ordonată. Sumatorul pe 8 biți.

2.1 Instrucțiunea *generate*

În lucrarea de laborator anterioară, la descrierea modelului structural al circuitului latch pe 8 biți, în cadrul arhitecturii acestuia s-au folosit 8 instrucțiuni de instanțiere pentru cele 8 instanțe ale componentei DFF. Această modalitate de descriere nu este, însă, convenabilă în cazul în care s-ar dori implementarea unui circuit asemănător, dar pe mai mulți biți, de exemplu un latch pe 32 de biți. După cum s-a văzut, structura circuitului latch pe 8 biți este o structură ordonată, în care aceeaşi componentă (DFF) este instanțiată de mai multe ori. Fiecare instanță a componentei DFF corespunde unui index din dimensiunea porturilor, de exemplu instanța cu eticheta F*i*, *i* = 0 ... 7, a componentei DFF este conectată la semnalele Pre, Clr, Clk, D(*i*), Q(*i*) şi QBar(*i*).

Limbajul VHDL oferă posibilitatea descrierii compacte a circuitelor având structuri ordonate de componente. Exemple de astfel de circuite: numărătoare, sumatoare, registre de deplasare, memorii, etc. Instrucțiunea prin care se realizează descrierea compactă a structurilor ordonate este instrucțiunea concurentă *generate*.

Instrucțiunea generate are două forme: forma cu *for*, respectiv forma cu *if*. Forma cu *if* se utilizează în cadrul unei instrucțiuni *generate* în forma cu *for*.

Sintaxa instrucțiunii *generate* în forma cu *for* este următoarea:

eticheta: **for** *index* **in** *domeniu* **generate**
    instrucțiuni concurente;
end generate;

unde *domeniu* reprezintă domeniul valorilor pentru *index*. Instrucțiunea *generate* determină repetarea instrucțiunilor concurente specificate în cadrul său pentru fiecare din valorile parametrului *index*. O instrucțiune generate în forma cu *for* poate conține alte instrucțiuni generate în forma cu *if*. Sintaxa instrucțiunii *generate* în forma cu *if* este următoarea:

eticheta: **if** *conditie* **generate**
    instrucțiuni concurente;
end generate;

Chiar dacă în cadrul unei instrucțiuni *generate* pot fi specificate diverse instrucțiuni concureente, uzual, se specifică instrucțiunea concurentă de instanțiere de componente, pentru descrierea structurală ordonată.

2.2 Sumatorul pe 8 biți.

În cele ce urmează se va realiza modelul structural al unui sumator pe 8 biți bazat pe sumatorul complet pe un bit (FullAdder) descris în unul din referatele de laborator anterioare.

Structura sumatorului pe 8 biți este prezentata in figura 1:

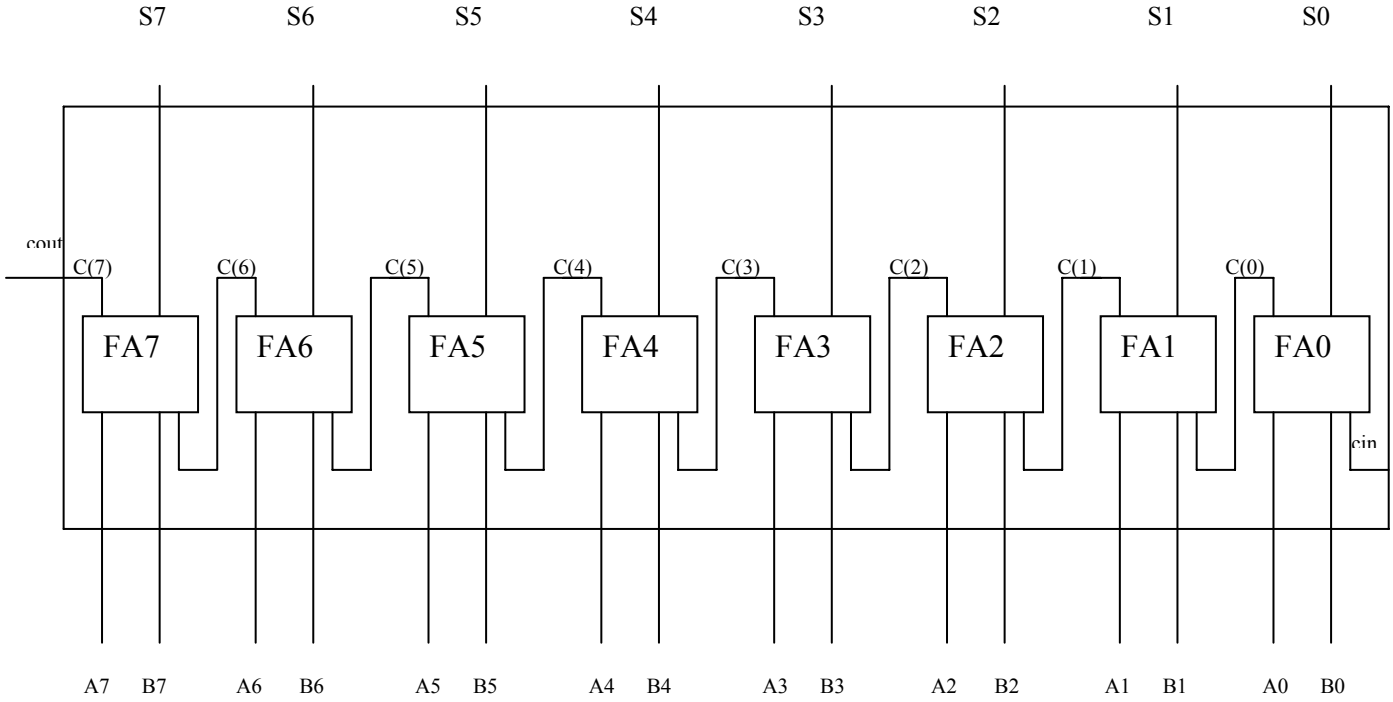

Figura 1 – *Structura sumatorului pe 8 biți*

Deoarece sumatorul pe 8 biți are o structură ordonată de instanțe ale sumatorului complet pe un bit mai jos este prezentată descrierea VHDL a acestuia folosind în cadrul arhitecturii instrucțiunea *generate*.

Declarația de entitate şi arhitectura asociată acesteia pentru sumatorul pe 8 biți (pe care îl denumim **Adder8**) sunt următoarele:

(fişier Adder8.vhd)

```
1   library IEEE;
2   use IEEE.std_logic_1164.all;

3   entity Adder8 is
4        port ( A,B : in Bit_Vector(7 downto 0);
5               Cin: in Bit;
6               Cout : out Bit;
7               Sum : out Bit_Vector ( 7 downto 0 ));
8   end  Adder8;

12  architecture Structure of Adder8 is
14     component FullAdder
15          port ( X,Y,Cin : in Bit;
17                 Cout,Sum: out Bit);
19     end component;
21     signal C: Bit_Vector (7 downto 0);
```

```
23  begin

25     Stages:
26           for i in 7 downto 0 generate
28                LowBit:
29                     if i = 0 generate
30                          FA0: FullAdder port map
31                               ( A(0), B(0), Cin, C(0), Sum(0));
32                     end generate;
34                OtherBits:
35                     if i /= 0 generate
36                          FA: FullAdder port map
37                               (A(i), B(i), C(i-1), C(i), Sum (i) );
38                     end generate;
40          end generate;
42    Cout <= C(7);

44  end;
```

Fără utilizarea specificației *generate* ar fi trebuit creată, pentru fiecare bit al sumatorului, o instrucțiune de instanțiere de componentă. Astfel, dimensiunea descrierii sumatorului (măsurată în numar de instrucțiuni) ar fi proporțională cu dimensiunea sa (masurată în numarul de biți).

Utilizarea instrucțiunii *generate* face ca dimensiunea descrierii să fie independentă de cea a sumatorului. De exemplu, pentru a construi un sumator pe 32 biți se înlocuieşte cifra ”7” din exemplu cu “31”.

În cadrul arhitecturii sumatorului, pentru a se putea utiliza instrucțiunea *generate* este necesar ca toate (sau majoritatea) instanțelor componentei FullAdder să fie conectate la semnale comune. Din acest motiv, toate semnalele interne care interconectează porturile Cin şi Cout ale fiecărui sumator pe un bit au fost grupate în cadrul unui singur semnal pe 8 biți numit C, declarat în linia 21.

Deoarece prima instanță (FA0) a componentei Fulladder are portul Cin conectat direct la portul Cin al Adder8, este descrisă separat față de celelalte instanțe. Acesa este motivul pentru care în descrierea de mai sus sunt utilizate ambele forme ale instrucțiunii *generate*, atât forma cu *for* cât şi forma cu *if*.

Astfel, în instrucțiunea *for ... generate* precedată de eticheta *“Stages”* în care parametrul *i* va fi variat în domeniul 0 … 7, sunt folosite alte două instrucțiuni *generate* în forma cu *if* pentru cazul $i = 0$, respectiv $i \neq 0$. Când $i = 0$ este instanțiată componenta FullAdder cu index 0, în timp ce pentru $i \neq 0$ sunt specificate compact instanțele componentei FullAdder cu indexul de la 1 la 7.

În finalul arhitecturii, deoarece ieşirea de transport a ultimului sumator pe un bit este considerată conectată la semnalul intern C(7), semnalului (portului) Cout i se atribuie semnalul C(7) (linia 42).

Observație: în arhitectura sumatorului pe 8 biți se poate renunța la utilizarea formei cu *if* a instrucțiunii *generate* dacă semnalul C este definit astfel încât să cuprindă şi semnalul de la intrarea Cin a instanței cu index 0.

În aceste condiții, liniile 25-40 vor fi înlocuite cu o descriere mai simplă de forma următoare:

```
Stages: for i in …… generate
      FA: FullAdder port map ( …)
end generate.
```

2.3 Testarea modelului sumatorului pe 8 biți.

Mai jos este prezentat un posibil circuit de test pentru sumatorul pe 8 biți:
(fişier Test_Adder8.vhd)

```
46  library IEEE;
47  use IEEE.std_logic_1164.all;

49  entity Test_Adder8 is end;

53  architecture Driver of Test_Adder8 is
55      component Adder8
56          port (A,B : in Bit_Vector (7 downto 0);
57                 Cin: in Bit;
58                   Cout : out Bit;
59                   Sum: out Bit_Vector ( 7 downto 0));

60      end component;

62      signal A,B,Sum:Bit_Vector (7 downto 0);
63      signal Cin,Cout:Bit :='0';
65  begin
67      UUT: Adder8 port map( A,B,Cin, Cout, Sum);
69      Stimulus:
70          process
71                variable Temp: Bit_Vector ( 7 downto 0);
72          begin
73                Temp :="00000000";
74                for i in 1 to 32 loop
75                      if i mod 2 /=1 then
76                            A <= Temp;
77                            B <= "00000001";
78                      else
79                            B <= Temp;
80                            A <= "00000001";
81                      end if;
82                      wait for 10 ns;
83                      Temp :=Sum;
84                end loop;
85                wait;
86          end process;
88  end;
```

În circuitul de test de mai sus modelul sumatorului pe 8 biți nu este testat pentru toate combinațiile de valori de la intrări deoarece sunt foarte multe (2^{17} combinații). Se poate aprecia dacă modelul este bun pe baza unui set mult mai mic de combinații. De exemplu, arhitectura circuitului de test de mai sus oferă posibilitatea de a simula şi verifica sumatorul pe 8 biți în cazul în care, pe una din intrări se aplică valoarea “00000001” iar pe cealaltă intrare se atribuie rezultatul sumei efectuate anterior.

Concret, sunt prevăzute 32 de iterații (cu instrucțiunea *for ... loop*) în care stabilirea valorilor atribuite celor două intrări este realizată în funcție de paritatea indexului *i*. Selectarea valorii pare sau impare a indexului *i* este realizată cu ajutorul instrucțiunii secvențiale *if* (liniile 75-81), care permite execuția condiționată a unor seturi de instrucțiuni secvențiale.

Forma generală a instrucțiunii *if* este următoarea:

```
if condiţie1 then
   instrucţiuni secvnţiale1
{elsif conditie2 then
   instrucţiuni secvnţiale2}
[else
   instrucţiuni secvnţiale3]
end if;
```

În cadrul formei generale acoladele "{…}" indică faptul ca textul din interiorul acestora poate fi multiplicat de zero sau mai multe ori. Parantezele "[…..]" arată ca textul este opţional.

Când o instrucţiune *if* este executată, condiţiile specificate în cadrul acesteia sunt verificate în ordinea intâlnirii lor, până este găsită cea adevărată. Sunt executate apoi instrucţiunile secvenţiale care corespund condiţiei adevărate.

Dacă nici o condiţie care urmează cuvintelor cheie **if** sau **elsif** nu este adevărată atunci, dacă este prezentă clauza **else**, sint executate instrucţiunile corespunzătoare acesteia.

În instrucţiunea *if* din arhitectura circuitului de test a sumatorului pe 8 biţi, în cadrul condiţiei este utilizat operatorul **mod** (modulo). Acest operator realizează următoarea funcţie aritmetică:

L mod R = L – (R*N),

unde operanzii L şi R sunt numere întregi iar N este cel mai mare întreg pentru care R*N ≤ L. Tipul rezultatului este acelaşi cu al operanzilor.

Astfel, dacă rezultatul operaţiei "*i mod 2* ", unde *i* parcurge bucla *for*, este diferit de 1 (*i* este par) atunci semnalul *A* va primi valoarea variabilei *Temp* iar B valoarea "00000001", în caz contrar (*i* este impar) valoarea variabilei *Temp* i se va atribui semnalului B iar semnalului A valoarea "00000001".

Pentru fiecare pas de execuţie a buclei *for … loop* variabila *Temp* va primi noua valoare calculată de sumator, prin portul *Sum*. De asemenea, la fiecare iteraţie procesul este suspendat timp de 10 ns. Aceasta presupune că simularea va trebui efectuată pe un interval de timp de minimum 320 ns pentru a parcurge toate cele 32 de iteraţii.

2.4. Aplicaţii 1

1. Copiaţi fişierele Adder8.vhd şi Test_Adder8.vhd în directorul de lucru VHDL.

2. Compilaţi fişierele de mai sus, efectuaţi simularea entităţii de test Test_Adder8 şi urmăriţi cu atenţie formele de undă a semnalelor pentru a aprecia dacă rezultatele simulării modelului sumatorului pe 8 biţi sunt corecte.

Observaţii: în fereastra *Wave* valorile semnalelor pe 8 biţi sunt reprezentate implicit în binar. Prin expandarea numelor acestor semnale (apăsare pe semnul + din dreptul acestora) se pot vizualiza formele de undă a fiecărui semnal component al celui pe 8 biţi.

Semnalele, în special cele pe mai mulţi biţi, pot fi vizualizate în fereastra *Wave* sub diverse forme (binar, zecimal, hexazecimal, etc). În cazul sumatorului pe 8 biţi este mai util să se vizualizeze semnalele multibit de la intrare şi ieşire în zecimal.

Pentru aceasta, în fereastra *Wave* selectaţi semnalele A, B şi Sum, apoi din meniu selectaţi:

Meniu ferestra Wave: **Format** > **Radix** > **Decimal**.

3. În fişierul Adder8.vhd, pe lângă arhitectura *Structure* realizaţi o nouă arhitectură numită Structure1 în care descrierea structurală să fie realizată compact doar cu instrucţiunea *generate* forma cu *for* (fără a se utiliza forma cu *if*).

4. Recompilaţi fişierul Adder8.vhd şi reluaţi simularea entităţii de test Test_Adder8.

Observaţie: Deoarece acum entitatea *Adder8* are două arhitecturi, *Structure* şi *Structure1* la simulare este necesar să se specifice care din aceste două arhitecturi este considerată pentru componenta Adder8. Pentru aceasta, în fişierul Test_Adder8.vhd, în partea declarativă a arhitecturii *Driver* se va adăuga o linie pentru configurarea componentei Full_Adder astfel:

```
for all: Adder8 use entity work.adder8(structure1);
```

5. Descrieti în limbaj VHDL latch-ul pe 8 biţi din lucrarea de laborator anterioară utilizând instructiunea *generate*.

3. Modelarea comportamentală. Modelul comportamental al unui registru de deplasare configurabil

3.1 Modelarea comportamentală. Principii generale

În VHDL modelarea comportamentală este reprezentată de instrucţiunea concurentă *process*. Instrucţiunea *process* sau procesul conţine un set de instrucţiuni care se execută secvenţial. O parte din instrucţiunile secvenţiale dintr-un proces sunt similare instrucţiunilor din limbajele de programare, cum ar fi limbajul C. De aceea, modelarea comportamentală se mai numeşte şi modelare de tip algoritm. Exemple de astfel de instrucţiuni sunt *if*, *case*, *for-loop*.

Instrucţiunea *process* utilizată la modelarea comportamentală a circutelor digitale are următoarea formă generală:

etichetă: **process** (*lista senzitivitate*)
declaraţii
begin
instrucţiuni secvenţiale
end process;

Lista senzitivitate este o listă de semnale (de exemplu, o parte din porturile de intrare ale circuitului) care determină activarea procesului şi, astfel, executarea instrucţiunilor secvenţiale din cadrul său.

În partea declarativă a instrucţiunii process se pot face diverse *declaraţii* ce pot fi folosite în procesul respectiv (de exemplu, declaraţii de tipuri de date, declaraţii de variabile, declaraţii de funcţii, etc).

Între **begin** şi **end** reprezintă zona instrucţiunilor secvenţiale. Aceste instrucţiuni sunt executate în ordinea în care sunt scrise. Câteva din cele mai utilizate instrucţiuni secvenţiale într-un proces sunt:

- instrucţiunea secvenţială de atribuire pentru semnale
- instrucţiunea de atribuire pentru variabile
- instrucţiunea *if*
- instrucţiunea *case*
- instrucţiunea *loop*

Orice proces se activează prima dată la momentul de timp t=0. După executarea instrucţiunilor procesul se suspendă până când apare un eveniment al oricărui semnal din lista de senzitivitate.

Observaţie: faţă de instrucţiunile process fără listă de senzitivitate (utilizate până în prezent în cadrul circuitelor de test) în cadrul cărora s-a utilizat instrucţiunea *wait* pentru suspendarea temporară sau definitivă a proceselor respective, la procesele cu lista de

senzitivitate NU se pote utiliza instrucţiunea *wait* pentru a suspenda procesele respective. Suspendarea/activarea proceselor de acest tip este dictată doar de semnalele din lista de senzitivitate.

3.2 Modelul comportamental al unui registru de depasare configurabil.

a) Specificaţii

În continuare se doreşte implementarea în VHDL a unui model comportamental pentru un registru de deplasare configurabil. Semnalul de intrare D are dimensiunea *m*, iar semnalul de iesire Q dimensiunea *n*, m<= n .

Tabela de adevăr a registrului de deplasare este următoarea:

CLK	CLR	LD	SH	DIR	Q
X	1	X	X	X	0
0	0	X	X	X	Q^*
1	0	X	X	X	Q^*
/	0	0	0	X	Q^*
∝	0	1	X	X	D
∝	0	0	1	0	0&Q^*(1 to n-1)
∝	0	0	1	1	Q^*(2 to n) & 0

Q^*-semnifică faptul că starea anterioară a registrului se regăseşte pe ieşiri

Din tabela de adevăr se poate observa că registrul de deplasare are o intrare *clear* (CLR) –de stergere asincronă, activă pe ‘1’ logic, care suprascrie toate intrarile. Incărcări sau deplasari pot apare doar pe un front crescător al semnalului de tact (CLK).

Daca semnalele LD si SH sunt in starea ‘0’ logic, la iesire nu are loc nici o modificare. Atunci cînd semnalul LD este in ‘1’ logic, valoarea curenta a semnalului D este încărcată în registru.

În schimb, daca LD este in ‘0’ logic iar SH este in ‘1’ logic atunci valoarea curentă a registrului este deplasată cu o poziţe la dreapta (daca semnalul DIR e in ‘0’ logic) sau la stânga (daca DIR e in ‘1’ logic).

Pozita rămasă vacantă în oricare din cazuri este inlocuita cu ‘0’.

b) Modelul comportamental în VHDL

Ma jos sunt prezentate declaraţia de entitate pentru registrul de deplasare configurabil şi arhitectura asociată acesteia:

(fisier ShiftN.vhd)

```
1   library IEEE;
2   use IEEE.std_logic_1164.all;

3   entity ShiftN is
4        port ( CLK : in Bit;
5               CLR : in Bit;
6               LD  : in Bit;
7               SH  : in Bit;
8               DIR : in Bit;
9               D   : in Bit_Vector;
10              Q   : out Bit_Vector);
12   begin
14       assert ( D'Length <= Q'Length )
15          report "Intrarea D nu poate fi mai mare decit iesirea Q"
16          severity Failure;
18  end ShiftN;
```

```
22  architecture Behavior of ShiftN is
23  begin
25    Shifter:
27      process (CLR, CLK)
29          subtype InBits is Natural range  D'Length-1 downto 0;
30          subtype OutBits is Natural range  Q'Length-1 downto 0;
31          variable State: Bit_Vector ( OutBits ) ;

33  begin
35      if CLR= '1' then
37          State := (others => '0' ) ;
38          Q <= State after 3 ns;
40      elsif CLK'Event and CLK = '1' then
42          if LD = '1' then
44                State := (others => '0');
45                State (InBits) := D;
46                Q <= State after 5ns;
48          elsif SH = '1' then

50                case DIR is
51                      when '0' =>
52                            State :='0' & State (State'Left downto 1);
53                      when '1' =>
54                            State := State (State'Left-1 downto 0) & '0'
;
55                end case;
56                Q <= State after 7ns;
58          end if;
60      end if;

62      end process;
64   end;
```

Descrierea de mai sus este o descriere comportamentală (*behavioral description*) deoarece modelul registrului de deplasare este descris ca algoritm şi nu ca structură fizică sau logică. În acest stadiu al proiectului este importantă doar corectitudinea algoritmului descris in limbaj VHDL.

Asa cum am mentionat anterior se propune posibilitatea configurarii numarului de biti din registrul de deplasare.

In VHDL, acest lucru este posibil prin utilizarea porturilor fara limitari (“*unconstrained ports”*).

Astfel semnalele “D” si “Q” sint declarate ca fiind de tip Bit_Vector, dar numarul elementelor nu este specificat, deci este nelimitat. Atunci cind am declarat aceleasi semnale pentru *latch* in lucrarea de laborator numarul 2 am precizat numarul elementelor.

Lipsa acestor limitari conduce la urmatoarele implicatii:

- modelul poate fi descris pentru a lucra cu un numar nelimitat de biti
- atunci cind entitatea este atribuita unei instante a unei componente, trebuie precizat numarul elementelor.

In interiorul declaratiei de entitate se verifica cu specificatia *assert* daca dimensiunea intrarii (*D*) nu este mai mica decit cea a iesirii (*Q*). Specificatia *assert* foloseste atributul predefinit “*Length*”, care furnizeaza numarul de elemente al tabloului.

Constructia *D’Length* va returna deci, numarul de elemente (biti) ale lui D. Deoarece acest nume nu este evaluat până cind entitatea ShiftN nu a fost atribuita unei componente, numarul de biti este cunoscut abia atunci.

Instructiunea *assert* prezintă clauza *severity*. Daca în urma evaluarii expresiei instructiunii *assert* rezultatul este fals atubci va fi afisat mesajul clauzei *severity*, dar si expresia *severity* insăşi.

Pentru a descrie comportamentul modelului sint utilizate instrucţiunea *process* si lista de sensitivitati ("*senesitivity list"*).

Lista de sensitivitati contine semnalele la care procesul este sensitiv. Un proces este sensitiv la un semnal când modificarea valorii semnalului produce executia procesului.

În consecinţă, acest proces va fi executat ori de cite ori semnalele "CLR " si "CLK" isi vor modifica valorile.

Specificatiile din interiorul procesului sînt executate în ordinea aparitiei lor (secvential). Apoi, procesul aşteaptă o nouă schimbare a valorii semnalelor la care este sensitiv. Orice proces este echivalent cu o instrucţiune concurentă, în consecinta, timpul de executie al procesului este 0 in timp simulat.

In interiorul procesului sunt două declaraţii de subtipuri. Subtipurile sînt limitări ale unor tipuri sau subtipuri existente.

Tipul predefinit *Integer* este declarat ca:

type Integer is range –2147483648 to 2147483647;

iar subtipul predefinit *Natural* este declarat ca fiind :

subtype Natural is Integer range 0 to 2147483647;

ceea ce înseamnă că subtipul *Natural* este un tip *Integer* nenegativ.

Astfel, subtipul *"InBits*" este un *Integer* , al cărui domeniu descrescător se întinde de la cel mai semnificativ bit al intrării *"D"* pînă la 0.

Similar, subtipul *"OutBits"* este un tip *Integer* al cărui domeniu descrescător se întinde de la cel mai semnificativ bit al ieşirii *"Q*" pînă la 0.

Starea internă a registrului de întîrziere este memorată în variabila *"State"* de tip *Bit_Vector* iar numărul de biţi este dat de subtipul *OutBits*. Aceasta înseamnă ca variabila "State" are aceeaşi dimensiune cu ieşirea "Q".

Linia 37 conţine specificatia de atribuire pentru a iniţializa variabila "State". Nu putem folosi un vector de zerouri deoarece nu ştim cîţi biţi conţine variabila "State" iar sintaxa limbajului VHDL impune ca numărul de biţi de ambele părţi ale specificatiei de atribuire să coincidă.

VHDL permite construirea de tipuri *Bit_Vector* (precum şi de alte tablouri) de orice lungime necesare specificatiei de atribuire utilizînd construcţia "*aggregate"*. Utilizînd un agregat care conţine asocierea *"others" (others association)* putem atribui variabilei "*State*" un vector de lungime egală cu numărul de zerouri.

Un agregat reprezintă o lista de valori ale unui tablou sau ale unui tip înregistrare. Este descris de o secventa de asociatii separate de virgula si inconjurate de paranteze.

In cazul nostru avem o singură valoare: *"others* =>0". Notaţia "*others* =>" indică faptul că elementele nemenţionate în agregat –aici toate- vor fi înlocuite cu valoarea de după săgeată.

La linia 40 clauza *elsif* utilizează atributul predefinit *"Event"*. Acest atribut care trebuie precedat de un semnal, returnează *"True"* la momentele de timp pentru care semnalul care il precede tocmai si-a schimbat valoarea, şi *"False"* altfel.

Astfel, constructia *"CLK'Event and CLK='1'"* semnifică următoarele: "*CLK* tocmai şi-a schimbat valoarea şi este în 1 logic". Din acest motiv expresia este adevărată exact pe frontul crescător al semnalului *"CLK"*.

In literatura de specialitate se recomanda utilizarea constructiei predefinite *rising_edge*(*CLK*) atunci cind semnalul *CLK* este de tip *std_logic, pentru a ne asigura ca tranzitia in '1' logic a semnalului de tact se face din '0' logic si nu din 'X', 'Z', '-' sau celelalte valori ale tipului std_logic*.

Totusi, cele doua modalitati folosite pentru detectarea frontului crescator al semnalului de tact sint similare la simulare, la sinteza logica insa, este posibila aparitia de rezultate diferite. Daca semnalul "CLK" este de tip *bit* cele doua constructii sint echivalente si la simulare si la sinteza logica.

In linia 45 datorită domeniilor descrescătoare cu limite identice la dreapta ale subtipurilor declarate la liniile 29 şi 30 sînt atribuiți biții cei mai puțin semnificativi ai variabilei *State*.

Liniile 52 şi 54 demonstrează utilizarea unui alt atribut predefinit "*Left"* care determină valoarea extremă stînga a indexului tabloului. In consecinta, atributul *"State'Left*" este identic cu "*Q'Length-1"*.

3.3 Testarea modelului comportamental

Programul de test prezentat mai jos instațiază o unitate supusă testului ShiftN având intrarea *D* pe 4 biți şi ieşirea *Q* pe 8 biți.

(fisier Test_ShiftN.vhd)

```
66  library IEEE;
67  use IEEE.std_logic_1164.all;

69  entity Test_ShiftN is end;

73  architecture Driver of Test_ShiftN is
75       component ShiftN
76          port ( CLK : in Bit;
77                 CLR : in Bit;
78                 LD : in Bit;
79                 SH : in Bit;
80                 DIR : in Bit;
81                 D : in Bit_Vector;
82                 Q : out Bit_Vector);
83       end component;

85      signal CLK, CLR,LD, SH, DIR : Bit ;
86        signal D : Bit_Vector ( 1 to 4) ;
87      signal Q : Bit_Vector ( 8 downto 1 );
89  begin
91      UUT : ShiftN port map ( CLK, CLR, LD, SH, DIR, D, Q );
93      Stimulus:
94          process
95          begin
97    --clear the register--
99                CLR <= '1' , '0' after 10ns;
100               wait for 10ns;
102   --load the register--
104               D <= "1110" ;
105               LD <= '1' , '0' after 10 ns;
106               CLK <= '0' , '1' after 3ns;
107               wait for 10ns;
109  -- left shift the pattern
111               SH <= '1' ;
112               DIR <= '1' ;
113               for i in 1 to 5 loop
114                     Clk <= '0' , '1' after 3 ns;
115                     wait for 10ns;
116               end loop;
```

```
118  --right shift the pattern
120               DIR <= '0' ;
121               for i in 1 to 8 loop
122                     CLK <= '0' , '1'  after 3ns;
123                     wait for 10ns;
124               end loop;
126         wait;
127         end process;
129  end;
```

Declarația de componentă nu impune limitări asupra porturilor *D* şi *Q*. Limitări sînt impuse la instanțierea componentei.

Semnalul *D* este declarat ca fiind de 4 biți în ordine crecătoare, semnalul *Q* este declarat ca fiind de 8 biți, în ordine descrescătoare *(8 downto 0)*. Cînd aceste semnale sînt asociate cu porturile registrului (linia 91) informația de limitare este transmisă modelului registrului cînd entitatea şi arhitectura sînt elaborate.

Reamintim că în model – în declarația subtipurilor "*InBits"* şi "*OutBits*" - am presupus că indexarea semnalelor *D* şi *Q* se va face descrescător (*"length-1 downto 0")*. Dar, în circuitul de test *Q* este declarat *"8 downto 0"* iar *D* de la 1 la 4.

Limbajul VHDL utilizează regula de aranjare a elementelor pentru rezolvarea acestor situații.

Atîta timp cît numărul de elemente din domenii este identic, elementul cel mai din stânga al unui domeniu este comparat cu cel mai din stînga din celălalt domeniu, următorul element dintr-un domeniu cu următorul din celălalt domeniu, şi aşa mai departe.

Astfel "D(1)" din circuitul de test este "D(3)" din model, "D(2)" din testbench este "D(2)" din model, "D(3)" din testbench este "D(1)" din model. Similar stau lucrurile şi în cazul semnalului Q.

3.4 Aplicatii 2

1. Copiați în directorul VHDL fişierele ShiftN.vhd şi Test_ShiftN.vhd
2. Compilați, efectuați simularea entității de test şi urmăriți cu atenție formele de undă rezultate în urma simulării.
3. Dezvoltati in limbaj VHDL modelul structural pentru circuitul secvential din figura 2. Lungimea *n* este o constantă specificată generic în declaratia de entitate.
4. Scrieti diagrama de stare a circuitului si specificati functia sa logica.
5. Creați un fişier cu numele Registru.vhd şi descrieți declarația de entitate şi arhitectura comportamentală a circuitului din fig. 2

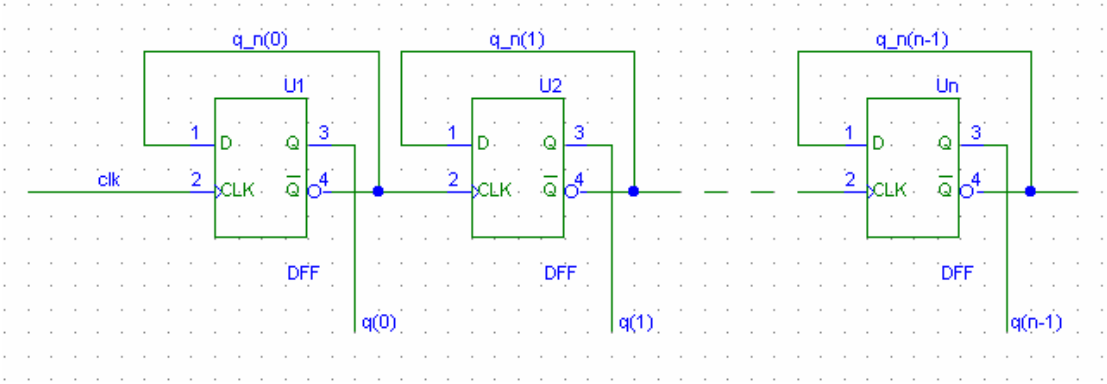

Fig.2 Circuit secvential