Lucrare Laborator Nr. 12
Modelarea mixtă în VHDL.
Implementarea unui algoritm de multiplicare.

1. Scopul lucrării

Însuşirea principiilor privind implementarea unui algoritm de multiplicare (înmulțire) în VHDL folosind modelarea mixtă. Este exemplificată implementarea unui algoritm de multiplicare cu adunare repetată şi deplasare-stânga dreapta folosind o serie de componente modelate şi testate în lucrările de laborator anterioare (sumatorul pe 8 biți, latch-ul pe 8 biți, registrul de deplasare configurabil şi circuitul de control modelat ca maşină cu stări finite). În ultima parte a lucrării este prezentat un circuit de test pentru verificarea multiplicatorului în care sunt folosite funcțiile *Convert* şi procedura *Clock* prezentate în lucrarea anterioară.

2. Algoritm de multiplicare cu adunare repetată şi deplasare stânga-dreapta

Înmulțirea a două numere binare fără semn pe N biți, A–multiplicand, B–multiplicator, conduce la un rezultat P (produs) pe 2N biți: P=A*B. Realizarea înmulțirii este similară cu cea din cazul înmulțirii numerelor zecimale. Astfel, bitul cel mai puțin semnificativ al multiplicatorului B se înmulțeşte cu biții multiplicandului A formând un produs parțial. La fel, se formează alte produse parțiale prin înmulțirea următorului bit al lui B cu toți biții lui A. La final, produsul este calculat ca suma a produselor parțiale, fiecare produs parțial fiind în prealabil deplasat spre stânga cu un număr de poziții egal cu indexul bitului corespunzător de la multiplicator, aşa cum se observă în exemplul următor.

Exemplu: Înmulțirea a două numere A şi B pe 4 biți. Aici B- multiplicand şi A– multiplicator.

```
B=0110; A=1011       B*A=01000010
```

```
    0110x
    1011
--------
    0110
   0110
 0110
--------
01000010
```

Există mai mulți algoritmi pentru înmulțirea a două numere reprezentate în format binar. Algoritmii clasici sunt următorii:
- algoritm de inmulțire cu deplasare dreapta;
- algoritm de inmulțire cu deplasare stânga;
- algoritm de inmulțire cu deplasare stânga-dreapta;
-algoritm de inmulțire Booth

În lucrarea de față se prezintă **algoritmul de înmulțire cu adunare repetată şi deplasare stânga-dreapta** din perspectiva implementării într-o structură logică. Acest algoritm constă în următorii paşi:

- Se încarcă numerele B şi A în regiştri
- P (produsul) este pe 2N biți. Inițial P=0.
- Repetă de N ori (sau până când toți biții din registrul A sunt 0):
    - Dacă LSB A = ‘1’
        - P<=P+B
    - Deplasează A cu o poziție la dreapta
    - Deplasează B cu o poziție la stânga

Mai jos este exemplificat detaliat modul de lucru al algoritmului pentru cazul înmulţirii numerelor B şi A pe 4 biţi din exemplul numeric anterior. Reg A şi Reg B reprezintă valorile pe 8 biţi de la ieşirea regiştrilor corespunzători numerelor B şi A. Valorile LSB ale Reg A în funcţie de care se realizează sumarea la iteraţia următoarea sunt reprezentate *italic*.

Exemplificarea iteraţiilor algoritmului:

	P	Reg B	Reg A
	----------	----------	----------
Initial	0000 0000	0000 0110	0000 101***1***
Iteraţia 1	0000 0000+		
	0000 0110		

	0000 0110	0000 1100	0000 010***1***
Iteraţia 2	0000 0110+		
	0000 1100		

	0001 0010	0001 1000	0000 001***0***
Iteraţia 3	0001 0010	0011 0000	0000 000***1***
Iteraţia 4	0001 0010+		
	0011 0000		

	0100 0010	0110 0000	0000 0000

3. Structura multiplicatorului pe 4 biţi

În figura 1 este prezentată structura internă a multiplicatorului care implementează algoritmul de inmulţire cu sumare repetată şi deplasare stânga-dreapta.

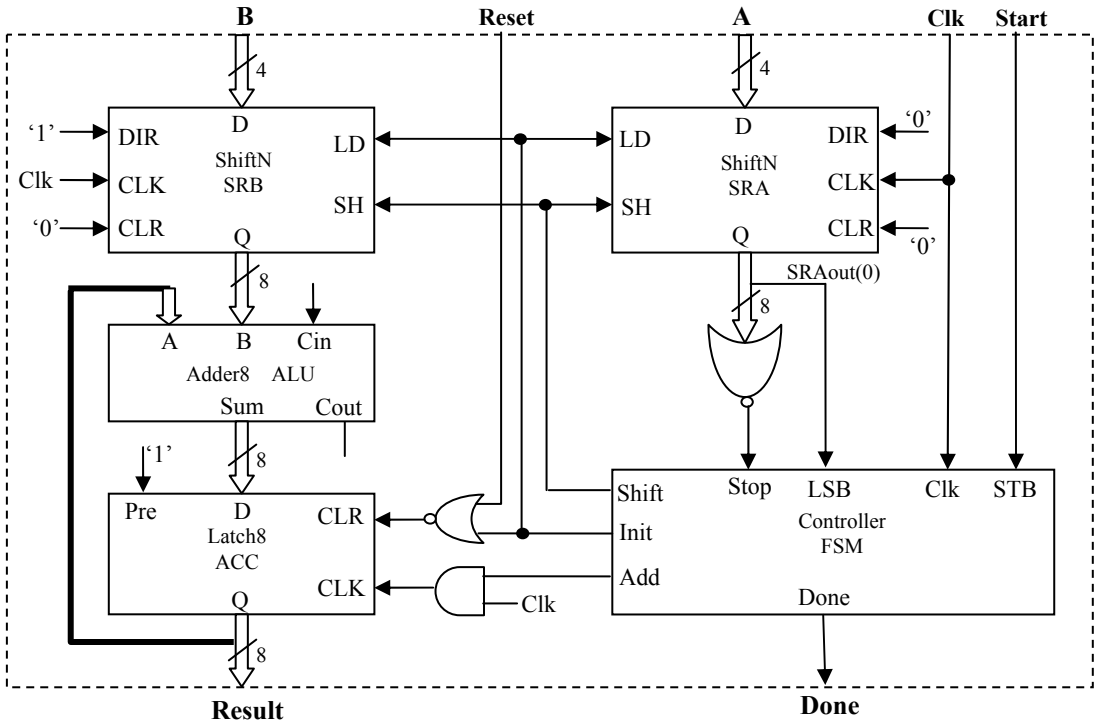

Figura 1 – *Schema bloc a multiplicatorului*

Circuitul de multiplicare are următoarele porturi:

- **Intrări**: *Clk*, *Start*, *Reset*, precum şi cei doi vectori de multiplicat *A* şi *B* pe 4 biţi.
- **Ieşiri**: *Result* – vector pe 8 biţi corespunzător produsului dintre intrările *A* şi *B*; *Done* – ieşire care indică sfârşitul procesului de multiplicare.

Circuitul de multiplicare constă din două registre de deplasare SRA şi SRB, câte unul pentru fiecare din intrările A şi B, un sumator pe 8 biţi notat ALU, un circuit de memorare şi acumulare tip latch pe 8 biţi notat ACC şi un circuit de comandă (controler), notat FSM, pentru controlul acestor componente. Pentru toate blocurile componente ale multiplicatorului vor fi folosite modelele prezentate şi testate în lucrările de laborator anterioare.

Când la intrarea *Start* este primit nivelul '1' logic, circuitul începe procesul pentru calculul produsului numerelor *A* şi *B* de la intrare. Astfel, după activarea intrării *Start* controlerul FSM generează un impuls *Init* care iniţializează (resetează) acumulatorul şi încarcă valorile intrărilor *A* şi *B* în cei doi regiştri de deplasare SRA şi SRB.

Dacă bitul cel mai puţin semnificativ de la ieşirea registrului SRA este '1' atunci controlerul FSM furnizează la ieşirea *ADD* un impuls care determină ca în acumulatorul ACC să se memoreze rezultatul sumei dintre valoarea de la ieşirea sa (*Result*) şi valoarea de la ieşirea registrului SRB.

După impulsul furnizat la ieşirea *ADD* controlerul furnizează un alt impuls la ieşirea *Shift*. Impulsul de la ieşirea *Shift* determină deplasarea cu o poziţie a biţilor de la registrele SRA şi SRB. În cazul registrului SRA deplasarea biţilor are loc **la dreapta** (de la MSB spre LSB) deoarece intrarea *DIR* a acestuia este conectată la '0' logic, în timp ce în cazul registrului SRB deplasarea biţilor are loc **la stânga** (de la LSB spre MSB) deoarece intrarea *DIR* a acestuia este conectată la '1' logic.

Dacă bitul LSB de la ieşirea Q a registrului SRA este '0', atunci nu trebuie să se mai sumeze ieşirea de la SRB cu cea a acumulatorului, ci controlerul FSM furnizează un impuls la ieşirea *Shift* prin care determină deplasarea cu încă o poziţie a biţilor de la registrele SRA şi SRB.

În final, dacă după deplasarea la dreapta toţi biţii de la ieşirea SRA sunt '0' atunci se consideră că procesul de multiplicare trebuie să se încheie. Când toţi cei 8 biţi de la ieşirea SRA sunt în '0' logic, funcţia logică NOR va determina '1' logic la intrarea *Stop* a controlerului. În acest caz controlerul va furniza un nivel de '1' la ieşirea *Done*, care indică sfârşitul procesului de multiplicare iar la ieşirea *Result* se află rezultatul final al multiplicării.

4. Modelul VHDL al multiplicatorului

În VHDL un multiplicator bazat pe un anumit algoritm de înmulţire poate fi descris comportamental ca o maşină cu stări finite. De exemplu, în cazul multiplicatorului prezentat în această lucrare, modelul comportamental poate fi dezvoltat pe baza modelului cu stări finite ale circuitului de control (FSM) la care se adaugă instrucţiuni pentru descrierea celorlalte blocuri.

În cele ce urmează se va prezenta modelul VHDL al multiplicatorului în concordanţă cu structura din figura 1. Astfel, pentru regiştrii de deplasare, sumatorul pe 8 biţi, acumulator şi circuitul de comandă se vor instanţia în arhitectura multiplicatorului componentele ShiftN, Adder8, Latch8 şi Controller a căror modele au fost studiate şi testate în lucrările anterioare. Pentru porţile logice NOR şi AND din figura 1 se vor utiliza expresii logice în instrucţuni de atribuire.

Pentru a instanția diverse componente într-o arhitectură, acestea trebuie, în prealabil, declarate. Declararea componentelor se poate face fie în partea declarativă a arhitecturii, fie în cadrul unui package.

În continuare se prezintă varianta în care declararea componentelor ce urmează a fi instanțiate în arhitectura multiplicatorului sunt declarate într-un package. Avantajul declarării într-un package este acela că declarațiile pot fi utilizate şi în cadrul altor proiecte.

4.1. Declararea componentelor într-un package

Package-ul care conține declarații ale componentelor ce vor fi instanțiate în arhitectura multiplicatorului este următorul:

(fişier Pack_Components.vhd)

```
3     package Components is
5        component Adder8
6           port (  A,B : in Bit_Vector(7 downto 0);
7                   Cin: in Bit;
8                   Cout : out Bit;
9                   Sum : out Bit_Vector ( 7 downto 0 ));
10       end  component ;

12       component Latch8
13          port ( D: in Bit_Vector (7 downto 0);
14                 Clk:in Bit;
15                 Pre: in Bit;
16                 Clr: in Bit;
17                 Q: out Bit_Vector ( 7 downto 0) );
18       end component ;

20       component ShiftN
21          port ( CLK : in Bit; CLR : in Bit;
23                 LD  : in Bit;  SH  : in Bit;  DIR : in Bit;
26                 D   : in Bit_Vector;
27                 Q   : out Bit_Vector
28                       );
29       end component;

31       component Controller
32          port ( STB : in Bit; CLK : in Bit;
33                 LSB : in Bit; Stop : in Bit;
34                 Init  : out Bit; Shift : out Bit;
35                 Add : out Bit; Done : out Bit );
40       end component;
42    end Components;
```

Toate componentele declarate în package-ul de mai sus au numele, lista de porturi şi tipul acestora identice cu cele ale entităților studiate anterior. În acest fel, în faza de elaborare a proiectului înainte de simulare se va asocia automat acestor componente perechile entitate-arhitectură corespunzătoare.

Pentru a face vizibil acest package în arhitectura multiplicatorului, se va adăuga clauza *use*

```
use work.components.all;
```

fie la începutul fişierului în care va fi descris multiplicatorul, fie în partea declarativă a arhitecturii acestuia. În acest referat clauza *use* va fi plasată în partea declarativă a arhitecturii.

4.2 Declarația de entitate şi arhitectura multiplicatorului

Fişierul cu declarația de entitate şi arhitectura mixtă a multiplicatorului este următorul: **(fişier Multiplier8.vhd)**

```
3    entity Mult8 is
4       port (A,B:    in  Bit_Vector(3 downto 0);
5             Start:  in  Bit;
6             CLK:    in  Bit;
7             Reset : in  Bit;
8             Result: out Bit_Vector(7 downto 0);
9             Done  : out Bit );
10   end Mult8;

14   architecture IterativeAdd of Mult8 is
16      use    work.Components .all;
18      signal SRAout, SRBout : Bit_Vector(7 downto 0);
19      signal ADDout, ACCout : Bit_Vector(7 downto 0);
20      signal Init, Shift, Add, Stop : Bit := '0' ;
22      signal High : Bit := '1';
23      signal Low  : Bit := '0';
24      signal OFL  : Bit;
26      signal ACCclk, ACCclr : Bit;
28   begin
30      ACCclr <= not (Init or Reset);
31      ACCclk <= CLK and Add;
32      Result <= ACCout;

34      SR_A: ShiftN port map
35          ( CLK => CLK,  CLR => Low,  LD => Init,  SH => Shift,
36            DIR => Low,  D => A,       Q => SRAout );

43      SRB: ShiftN port map
44          ( CLK => CLK,  CLR => Low,  LD => Init,  SH => Shift,
48            DIR => High, D => B,       Q => SRBout );

52      ALU: Adder8 port map
53          ( ACCout, SRBout, Cin => Low, Cout  => OFL, Sum => ADDout);

55      ACC: Latch8 port map
56          ( ADDout, Q=>ACCout, Pre=>High, Clk=>ACCclk, Clr=>ACCclr);

58      CHK: Stop <= not( SRAout(7) or SRAout(6) or SRAout(5) or SRAout(4) or
59                    SRAout(3) or SRAout(2) or SRAout(1) or SRAout(0) );

61      FSM: Controller port map
62          ( Start, CLK, SRAout(0), Stop, Init, Shift, Add, Done);
64   end;
```

Aşa cum se observă din descrierea de mai sus, în partea declarativă a arhitecturii nu sunt prezente declarații ale componentelor, ci este utilizată clauza „use” (linia 16) pentru a face vizibil în cadrul arhitecturii toate declarațiile de componente din package-ul Components.

În cadrul descrierii multiplicatorului semnalul *Result* este declarat la linia 8 ca fiind port de ieşire. Portul *Q* al componentei Latch8 este conectat la semnalul intern *ACCout* declarat la linia 19. Valorile acestui semnal se atribuie portului de ieşire *Result* (linia 32).

Corpul arhitecturii utilizează specificații de atribuire pentru semnale (liniile 30-32 şi 58-59) pentru a descrie o parte a circuitului şi instrucțiuni de instanțiere de componente. Posibilitatea de a utiliza stiluri de descriere diferite în aceeaşi arhitectură reprezinta unul din avantajele utilizarii limbajului VHDL. În general, într-o arhitectură pot fi prezente simultan două sau chiar toate din cele trei instrucțiuni (atribuire pentru semnale, instanțiere de componente, process) care definesc cele trei stiluri de modelare (data-flow, structural şi comportamental).

Instrucțiunile de instanțiere de componente prezente în arhitectura multiplicatorului introduc una din capabilitățile VHDL privind modul de asociere a porturilor componentelor la semnale. Astfel, la instanțierea controlerului (liniile 61-62) se asociază porturile componentei cu semnalele din arhitectura multiplicatorului utilizând *asocierea poziţională*.

La asocierea poziţională, primul semnal din lista după specificația **port map** corespunde primului port al componentei, al doilea semnal cu al doilea port, ş.a.m.d .

Pentru controler putem construi următorul tabel cu asocieri porturi-semnale:

Porturi controler	**STB**	**CLK**	**LSB**	**Stop**	**Init**	**Shift**	**Add**	**Done**
Semnale multiplicator	**Start**	**CLK**	**SRAout(0)**	**Stop**	**Init**	**Shift**	**Add**	**Done**

Asocierile porturi-semnale pot fi specificate şi explicit utilizând *asocierea numelor*. De exemplu, la cele două instanțieri cu etichetele SR_A şi SRB a registrului de deplasare (liniile 34-48) fiecare asociere în lista după specificația **port map** este de forma "port componentă => semnal multiplicator".

Totodată, este posibilă combinarea dintre asocierea poziţională şi cea a numelor în aceeaşi listă de asocieri aşa cum este în cazul intanțierii componentei Adder8 (liniile 52 şi 53).

Toate asocierile poziționale, dacă există, trebuie să preceadă asocierea numelor. În caz contrar, compilatorul VHDL nu poate stabili corect asocierea dintre porturi şi semnale.

În linia 34, la instrucțiunea de instanțiere a componentei ShiftN s-a utilizat eticheta "SR_A" în loc de "SRA" aşa cum este notat în figura 1, deoarece "sra" este cuvânt rezervat al VHDL şi face parte din categoria operatorilor de shiftare-rotiere (sra – shift right arithmetical).

În figura 1 se observă că portul Cout al componentei Adder8 rămâne neconectat. Totuşi, în arhitectura multiplicatorului acest port este conectat la semnalul intern OFL (linia 53) declarat special pentru acest scop. În general, la instanțierea unei componente, un port neconectat se poate specifica în lista de asociere folosind în loc de nume de semnal cuvântul "open". Astfel, în linia 53 se poate scrie:

```
Cout  => open
```

Pe de altă parte, dacă un port de intrare a unei componente trebuie conectat la un semnal de nivel logic constant (de exemplu, '0' sau '1'), fie se declară în arhitectură semnale interne cărora li se atribuie valoarea logică dorită, fie la instanțierea componentei pentru portul respectiv se asociază valoarea logică în loc de nume de semnal.

Astfel, în arhitectura multiplicatorului în liniile 22 şi 23 s-au declarat semnalele *High* şi *Low* cu valorile '1', respective '0'. Ca urmare toate porturile care trebuie conectate la unul din aceste două stări sunt asociate cu unul din aceste două semnale. De exemplu, portul *DIR* al registrului SRB este conectat la semnalul *High*.

În loc de utilizarea semnalelor Low şi High, se poate specifica direct în lista de asociere valoarea logică corespunzătoare. De exemplu, instanța SRB a registrului ShiftN poate fi specificată astfel:

```
SRB: ShiftN port map
      ( CLK => CLK,  CLR => '0',  LD => Init,  SH => Shift,
        DIR => '1',    D => B,           Q => SRBout );
```

5. Testarea multiplicatorului

În fişierul Test_Multiplier8.vhd este prezentat un program de test pentru multiplicator.
(fisier Test_Multiplier8.vhd)

```
69   entity Test_Mult8 is end;

73   architecture Driver of Test_Mult8 is
75      component Mult8
76          port (A,B   : in  Bit_Vector(3 downto 0);
77                Start : in  Bit;
78                CLK   : in  Bit;
79                Reset : in  Bit;
80                Result: out Bit_Vector(7 downto 0);
81                Done  : out Bit );
82      end component;
84      signal A,B     : Bit_Vector(3 downto 0);
85      signal Start, Done : Bit := '0' ;
86      signal CLK     : Bit ;
87      signal Reset   : Bit;
88      signal Result  : Bit_Vector ( 7 downto 0);
90      signal DisplayA, DisplayB, DisplayResult : Natural ;
92      use    work.Utils.all;
94   begin
96      C:   Clock (CLK, 10 ns, 10 ns);
98      UUT: Mult8 port map (A,B,Start,CLK,Reset,Result, Done);
100     Reset <= '1' ,'0' after 1ns;
102     Stimulus:
103         process
104         begin
105               for i in 1 to 3 loop
106                     for j in 4 to 7 loop
107                           DisplayA <= i;
108                           DisplayB <= j;
109                           A <= Convert (i, A'Length );
110                           B <= Convert (j, B'length);
111                           wait until CLK'Event and CLK='1';
112                           Start <='1', '0' after 20ns;
113                           wait until Done ='1';
114                           DisplayResult <= Convert(Result);
115                           wait until CLK'Event and CLK = '1';
116                     end loop;
117               end loop;
118         wait;
119         end process;
121  end;
```

În cadrul arhitecturii de test, la linia 96, este apelată procedura "Clock" definită în package-ul *Utils* din lucrarea de laborator anterioară. Aici procedura este apelată ca instrucțiune concurentă deoarece apare în corpul arhitecturii, şi nu în corpul unui process. Procedura este folosită la generarea semnalului de tact pentru multiplicator şi testbench.

Procesul pentru generarea de stimuli (liniile 102-119) utilizează două bucle imbricate pentru a testa multiplicatorul în cazul în care la porturile de intrare *A* şi *B* se aplică pe rând valorile {1,2,3} respectiv {4,5,6,7}.

În interiorul buclei valorile de multiplicat sunt aplicate mai întâi semnalelor "DisplayA" şi "DisplayB" pentru a putea fi afişate ca numere întregi (liniile 107, 108), apoi sunt convertite în tipuri Bit_Vector. Specificațiile de la liniile (109,110) apelează secvențial (în corpul procesului) funcția *Conver*t corespunzătoare din pachetul *Utils* şi atribuie rezultatele conversiilor intrărilor multiplicatorului.

Specificația *wait* de la linia 111 semnalizează apariția unui front crescător al semnalului de tact (CLK). Condiția *until* este cea mai generală formă de expresie senzitivă la un eveniment.

După apariția frontului crescător al semnalului *CLK*, semnalului *Start* i se atribuie valoarea '1' timp de 20 ns (linia 112), ceea ce permite inițializarea multiplicatorului şi începerea procesului de calcul al produsului.

Următoarea specificație *wait* (linia 113) determină suspendarea procesului până când multiplicatorul termină de efectuat înmulțirea, moment semnalizat de acesta prin trecerea în '1' logic a ieşirii *Done*.

După finalizarea înmulțirii, rezultatul de tip Bit_Vector furnizat la portul şi semnalul *Result* este apoi convertit la un tip întreg prin apelarea celei de-a doua funcții *Convert* din package-ul *Utils* şi atribuit semnalului *DisplayResult*.

Specificația *wait* de la linia 115 suspendă procesul până la următorul front crescător al semnalului de ceas pentru a permite vizualizarea rezultatului de la fiecare iterație.

5.1 Rezultatele simulării multiplicatorului

În figura 2 sunt prezentate formele de undă sau valorile semnalelor obținute în urma simulării entității de test Test_Mult8 pentru validarea modelului VHDL al multiplicatorului descris de entitatea *Mult8* şi arhitectura *IterativeAdd.* Se observă că din momentul în care semnalul *start* ia valoarea 1 (începutul efectuării înmulțirii) sunt necesare câteva perioade ale semnalului de ceas (*clk*) până când semnalul *done* indică terminarea calculului.

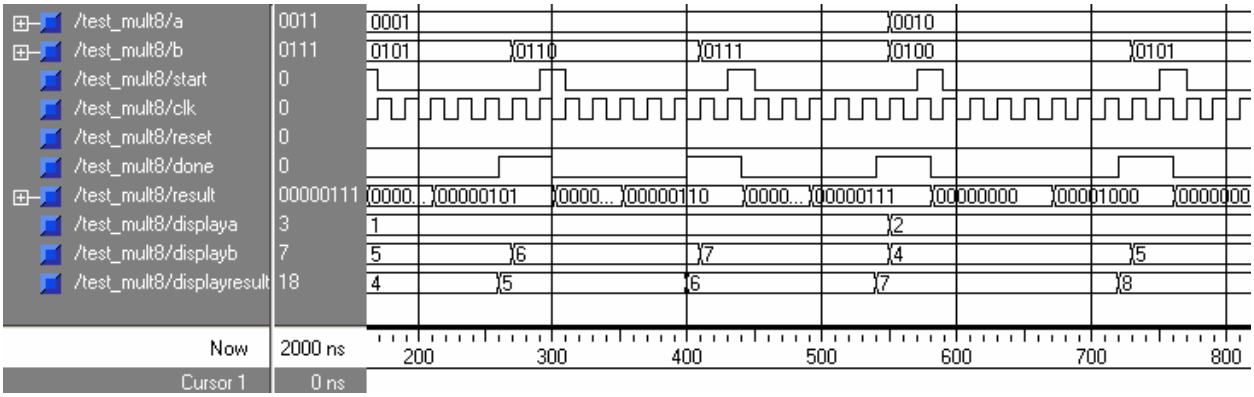

Figura 2 – *Rezultatele obținute cu arhitectura Driver a entității Test_Mult8*

6. Aplicații

1. Efectuați pe o foaie de hârtie înmulțirea dintre următoarele două numere binare pe 6 biți, X = 101001 şi Y = 110011 şi descrieți valorile produsului P şi a regiştrilor Reg X şi Reg Y corespunzătoare iterațiilor algoritmului de înmulțire cu adunare repetată şi deplasare stânga-dreapta, similar ca la pct. 2 din referat.

2. Urmăriți figura 1 şi explicațiile de la pct. 3 privind modul de lucru al multiplicatorului pe 4 biți. Pentru aprofundarea înțelegerii modului în care funcționează şi sunt comandate componentele din cadrul multiplicatorului urmăriți în paralel diagrama de stări a controlerului.

3. Copiați în directorul de lucru fişierele Multiplier8.vhd, Test_Multiplier8.vhd şi Pack_Components.vhd. Compilați aceste fişiere şi corectați eventualele erori

Observație: Deoarece entitatea Mult8 a multiplicatorului conține în structura sa instanțe ale componentelor studiate în lucrările de laborator anterioare şi deoarece unele din acestea au structura alcătuită din alte componente (de exemplu, sumatorul Adder8 conține 8 instanțe ale entității Full_Adder), dacă este cazul, recompilați toate fişierele corespunzătoare entităților din structura multiplicatorului. În Figura 3 sunt prezentate toate componentele care intră în structura multiplicatorului precum şi modul lor de ierarhizare.

4. Efectuați simularea entității de test *Test_Mult8* pe durata a 2us. Vizualizați formele de undă şi valorile semnalelor din entitatea de test similar ca în figura 2. Urmăriți dacă rezultatele sunt corecte pentru toate valorile aplicate la cele două intrări, A şi B.

5. În fereastra Wave determinați durata şi numărul de perioade de ceas necesare pentru efectuarea înmulțirii pentru fiecare din valorile intrării A: A=1, A=2 şi A=3. Duratele se vor determina dintre momentele frontului pozitiv al semnalului *start* şi momentele frontului pozitiv al semnalului *done*. Pentru determinarea duratelor utilizați butoanele pentru afişarea şi deplasarea cursoarelor:  Explicați de ce duratele pentru efectuarea înmulțirii sunt diferite în cele trei cazuri. Durata înmulțirii depinde de valoarea de la intrarea B?

6. Având încărcată entitatea Test_Mult8 în simulator, din fereastra principală a programului ModelSim deschideți fereastra **Structure** (selectați **View** -> **Structure**) şi vizualizați structura ierarhică a entității de test precum şi a multiplicatorului. Se va apăsa pe butonul + pentru a vizualiza structura fiecărei componente. Urmăriți structura din ferestra **Structure** în comparație cu structura ierarhică prezentată în figura 3.

7. Fie următoarea descriere simplă în VHDL pentru un multiplicator pe 8 biți:

```
library ieee;
use ieee.std_logic_1164.all;

entity Mult16 is
  port (A, B: in std_logic_vector (7 downto 0);
          Y:    out  std_logic_vector (15 downto 0));
end Mult16;

architecture Comp of Mult16 is
begin
   Y<= A * B;
end Comp;
```

Care alt package trebuie făcut vizibil cu clauza *use* pentru a putea utiliza operatorul "*" în arhitectura acestui model? Editați şi compilați fişierul pentru a verifica răspunsul.

8. *Temă pentru acasă*: Imaginati şi descrieți un alt program de test pentru entitatea Mult8 utilizând, de exemplu, metoda de citire a vectorilor de test dintr-un tablou, similar cum s-a procedat în cazul testării sumatorului complet pe un bit Full_Adder.

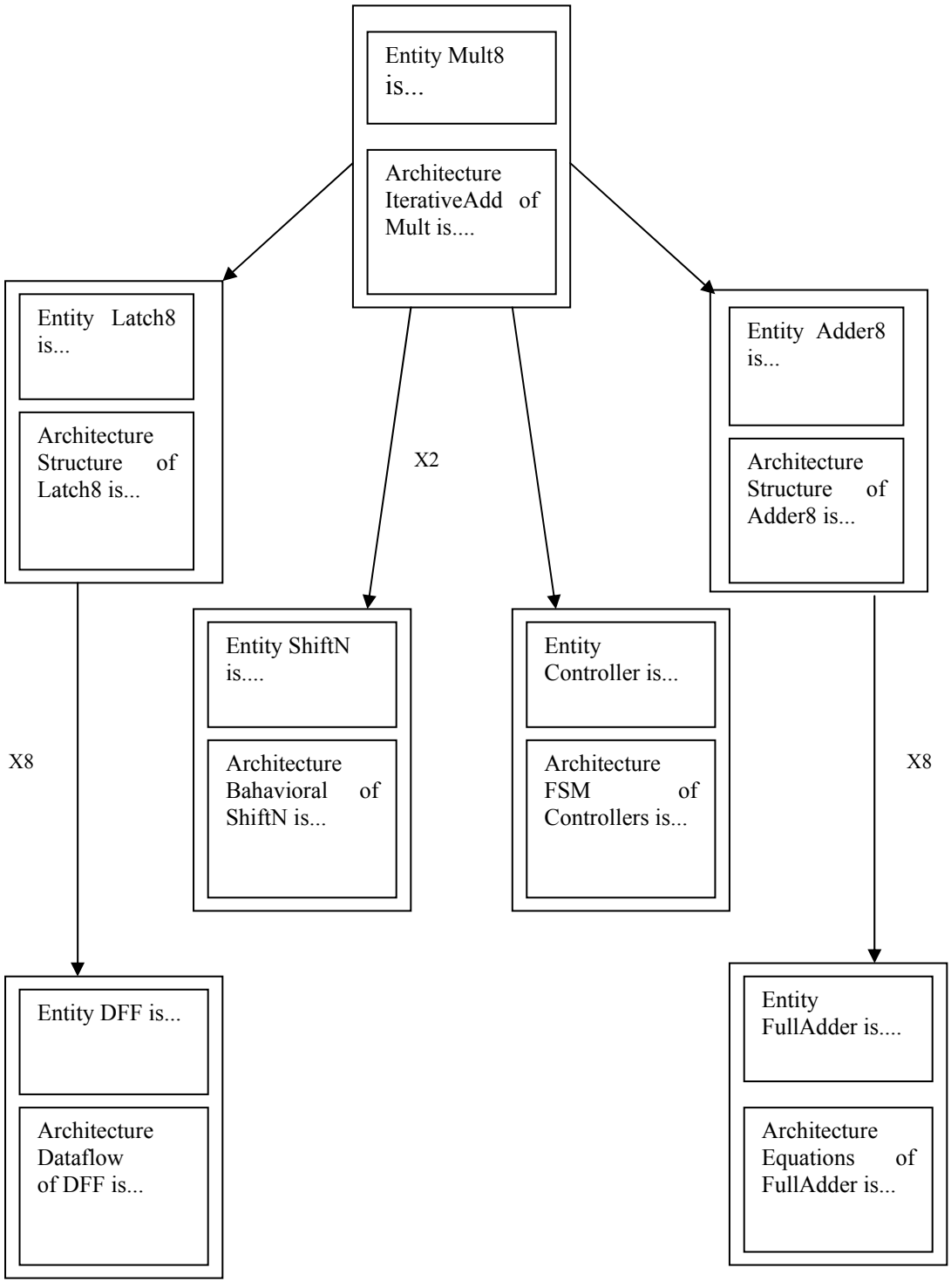

Figura 3 – *Structura ierarhică a componentelor din cadrul multiplicatorului Mult8*