Organizarea Calculatoarelor
LABORATOR 5 – Implementarea şi simularea unui sumator/scăzător pe 4 biţi în VHDL

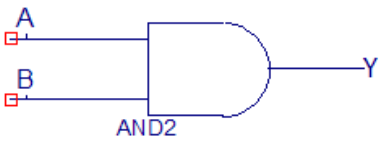
SCOPUL LUCRĂRII

Sumator/scăzătorul din laboratorul precedent se va descrie folosind limbajul VHDL iar apoi se va simula.

Chestiuni teoretice

Metodologia de proiectare bazată pe scheme prezintă un mare inconvenient: **se pierde foarte mult timp cu desenarea**. Chiar dacă sumator/scăzătorul din laboratorul anterior efectua operaţii numai pe patru biţi, desenarea lui a durat 1 oră. Imaginaţi-va cum ar fi să-l desenaţi pe 32 de biţi! Cum ar fi să proiectăm un întreg procesor cu scheme?

Specificarea funcţionalităţii s-ar face mult mai repede dacă, de exemplu, în loc de a desena o poarta AND:



am scrie **Y<= A and B;**

Evident **Y<= A and B;** se scrie mult mai repede decât se desenează.

În limbajele de descriere hardware (HDL) firele prin care se conectează elementele logice de prelucrare sunt obiecte din clasa **wire** în Verilog sau **signal** in VHDL. În continuare vom folosi limbajul VHDL.

Informaţia vehiculată prin intermediul semnalelor poate fi de mai multe feluri. În continuare se va folosi numai tipul **std_logic**. **Acest tip de date este o enumerare alcătuită din 9 caractere.** Cele 9 caractere surprind toate situaţiile care pot apare în proiectarea logică. Trei dintre aceste simboluri sunt: ‚1' pentru unu logic, ‚0' pentru zero logic şi ‚Z' pentru tristate.

Aşa cum s-a văzut în laboratorul precedent, pentru a simplifica schema mai multe conexiuni se pot specifica prin intermediul elementelor de tip vectorial, ca de exemplu magistrala. Elementele de tip vectorial sunt tot semnale dar tipul lor este **std_logic_vector**. Dimensiunea vectorului se specifică în paranteze, imediat după cu cuvântul cheie **std_logic_vector**.

În limbajul C atunci când se defineşte un vector în declaraţie se specifică lungime vectorului. Componentele vectorului sunt definite automat: indicele cel mai mic este zero iar cel mai mare este lungimea minus 1. De exemplu:

```
char my_string[4];
```

defineşte un vector de caractere cu 4 elemente, primul fiind `my_string[0]` iar ultimul `my_string[3]`.

În VHDL se specifică elementul cel mai din stânga şi elementul cel mai din dreapta. Exista două posibilităţi: indexul elementului cel mai din stânga este mai mic sau egal cu indexul elementului cel mai din dreapta sau invers. De exemplu:

```
signal my_vect       :  std_logic_vector(1 to 4);
```

defineşte un vector cu 4 elemente. De la stânga la dreapta acestea sunt *my_vect*(1), *my_vect*(2), *my_vect*(3) şi *my_vect*(4).

Prin declaraţia:

```
signal your_vect     :  std_logic_vector(7 downto 5);
```

defineşte un vector cu 3 elemente. De la stânga la dreapta acestea sunt *your_vect*(7), *your_vect*(6) şi *your_vect* (5).

Spre deosebire de limbajul C limitele vectorului nu sunt predefinite şi în plus apare noţiunea de gamă: (1 **to** 4) este o gamă crescătoare iar (7 **downto** 5) este o gamă descrescătoare.

Indicii vectorilor de tip **std_logic_vector** sunt obligatoriu numere naturale.

Alte elementele ale limbajului VHDL vor fi reamintite pe măsură ce se va implementa proiectul, prin analogie cu reprezentarea de tip schematic.

Desfăşurarea lucrării

Pasul 1: Crearea proiectului.

Se face conform metodologiei prezentate în primul laborator. Deşi vom lucra mai mult în VHDL, proiectul va fi tot de tip **schematic**. Proiectul se va numi AddSub4V.

Pasul 2: Crearea fişierului VHDL în care se va descrie sumator/scăzătorul pe 4 biţi.

În figura 1 este prezentată schema sumator-scăzătorului pe 4 biţi desenată în laboratorul precedent.

NU DESENATI ACEASTA SCHEMA, ATI FACUT-O DEJA!

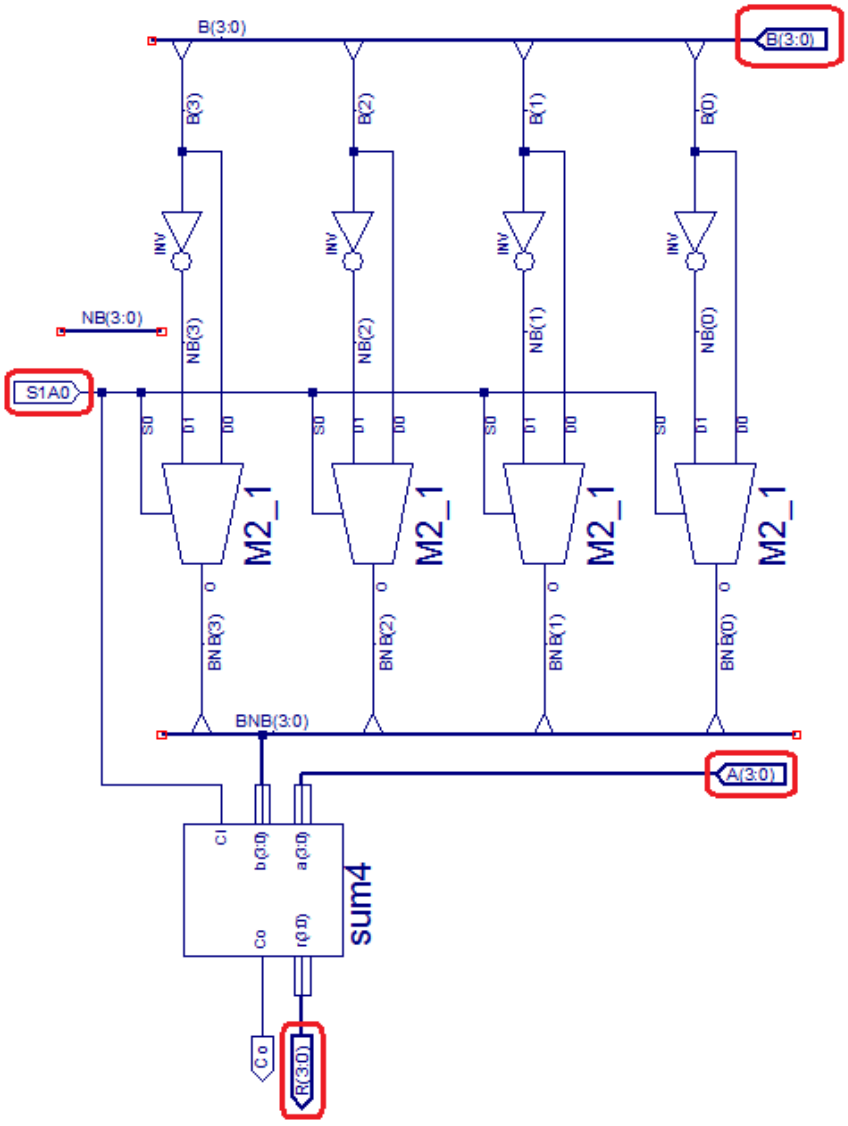


figura 1

În VHDL atât simbolul cât şi funcţionalitatea sunt descrise într-un singur fişier. Descrierea simbolului se face prin intermediul unui element al limbajului numit **entity** iar funcţionalitatea printr-un alt element numit **architecture**. Mediul ISE construieşte automat scheletul fişierului VHD, schelet ce conţine entitatea şi arhitectura.

Mai mult, prin intermediul unei ferestre de dialog se pot specifica complet elementele VHDL echivalente cu porturile (I/O Marker în Xilinx) din schemă. Pentru sumator/scăzător acestea sunt: **A(3:0)**, **B(3:0)**, **S1A0** şi **R(3:0)**. Vezi marcajele cu roşu din figura 1. Pentru moment ieşirea **Co nu se va mai implementa** deoarece necesită cunoştinţe suplimentare de VHDL.

Adăugaţi la proiect un nou fişier. Numele acestui fişier este **as4v** iar tipul său este **VHDL Module**. Ca urmare va apare o fereastră de dialog ce permite specificarea semnalelor prin care as4v se va conecta cu ale simboluri. Informaţiile despre porturile (marcherii IO) **A(3:0)**, **B(3:0)**, **S1A0** şi **R(3:0)** se introduc în această fereastră. În urma acestei operaţii se obţine situaţia din figura 2.

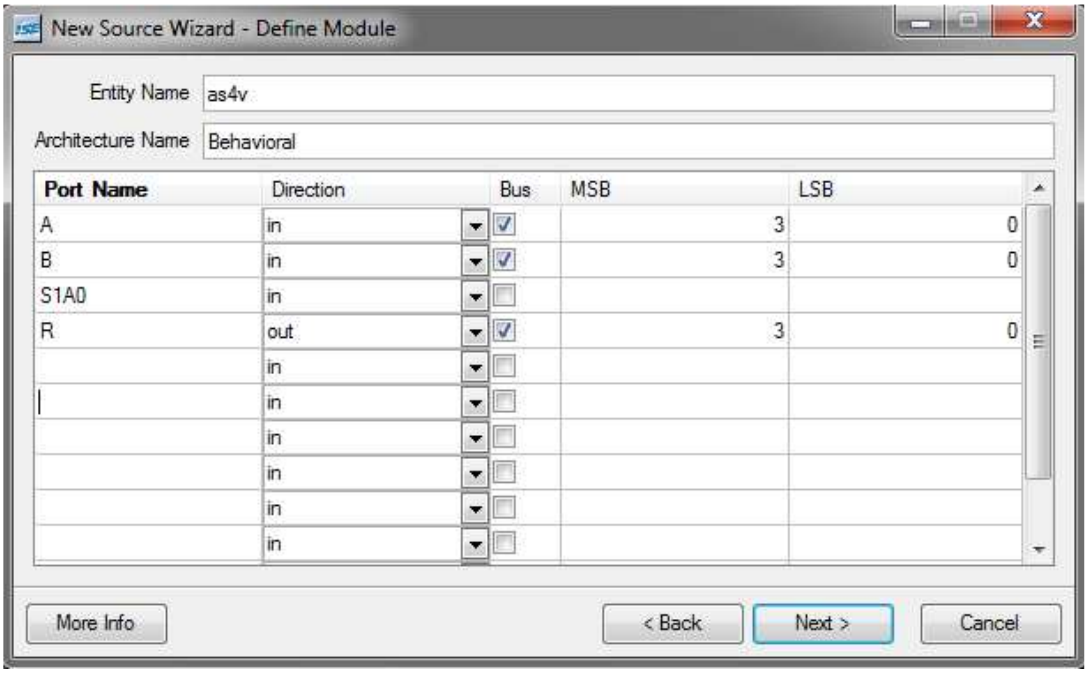


figura 2

După ce apăsaţi **Next** şi apoi **OK**, ISE va genera automat fişierul as4v.vhd şi îl va adăuga la proiect. O parte din acest fişier este prezentată în figura 3:

```
20  library IEEE;
21  use IEEE.STD_LOGIC_1164.ALL;
22  use IEEE.STD_LOGIC_ARITH.ALL;
23  use IEEE.STD_LOGIC_UNSIGNED.ALL;
24
25  ---- Uncomment the following library declaration if instantiating
26  ---- any Xilinx primitives in this code.
27  --library UNISIM;
28  --use UNISIM.VComponents.all;
29
30  entity as4v is
31     Port(
32        A      : in  STD_LOGIC_VECTOR (3 downto 0);
33        B      : in  STD_LOGIC_VECTOR (3 downto 0);
34        S1A0   : in  STD_LOGIC;
35        R      : out STD_LOGIC_VECTOR (3 downto 0)
36     );
37  end as4v;
38
39  architecture Behavioral of as4v is
40
41  begin
42
43
44  end Behavioral;
```

figura 3

Atenţie: nu veți obține exact aspectul din figura 3. Elementele definite la nivelul entității au fost indentate manual pentru a fi identificate mai uşor..

Aceste elementele sunt uşor de interpretat. Entitatea este definită cu prin cuvintele:

```
entity nume_entitate is
.....
end [nume_entitate] ;
```

nume_entitate după **end** este opţional.

Pinii (terminalele) entităţii se definesc în interiorul structurii **port**:

```
port(
      port_1;
      port_2;
      …..
      port_n-1;
      Port_n
 );
```

Un port se defineşte astfel:

```
Nume_port : in | out | inout tip_port;
```

Simbolul ‚|' înseamnă „sau", în descrierea BNF.

În fişierul din figura 3 semnalele **A(3:0)**, **B(3:0)** şi **S1A0** sunt semnale de interfaţă; direcţia lor este **in** iar tipul este **std_logic_vector** cu 4 elemente, gamă descrescătoare. Semnalul **R(3:0)** este ultimul semnal de interfaţă. Direcţia sa este **out** iar tipul sau este **std_logic_vector** cu 4 elemente, gamă descrescătoare.

De remarcat că după ultima declaraţie de port nu se pune punct şi virgulă. De asemenea în declaraţia semnalelor de interfaţă nu apare cuvântul cheie **signal**. Deoarece interfaţarea modulelor nu se poate face decât prin semnale, **signal** ar fi în acest caz redundant.

Pasul 3: Descrierea celor 4 inversoarelor

După ce se defineşte modul de interfaţare al modulului se trece la specificarea funcţionalităţii acestuia. Această operaţie se face la nivelul arhitecturii, după cuvântul cheie **begin**. Mai întâi vom descrie cele 4 inversoare. În figura 4 sunt prezentate detaliat cele 4 inversoare din figura 1.

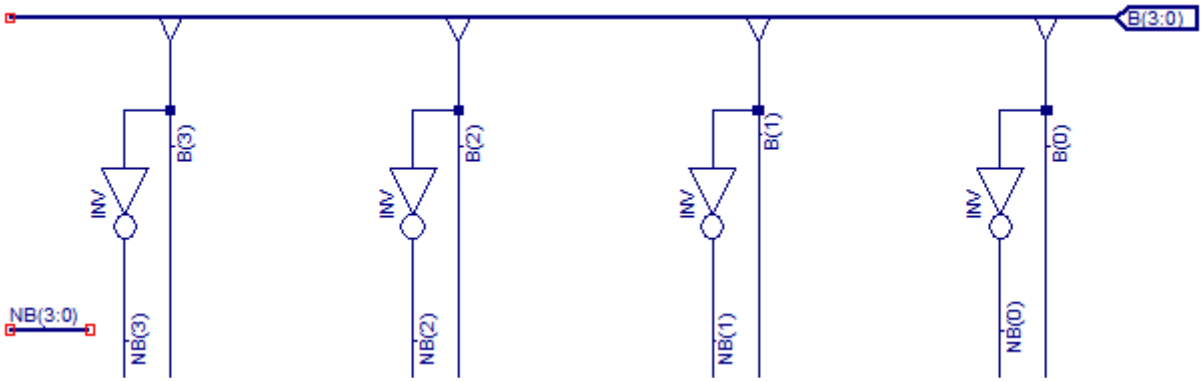


figura 4

În cazul celor patru inversoare ar trebuie să scriem în VHDL:

```
NB(0) <= not B(0);
NB(1) <= not B(1);
NB(2) <= not B(2);
NB(3) <= not B(3);
```

Deşi este mai uşor decât sa desenăm 4 inversoare, operaţia de specificare devine consumatoare de timp pentru 32 de inversoare. Din fericire această problemă s-a rezolvat de mult cu ajutorul ciclului **for**.

În C am scrie:

```
for(k=0; k<4; k++){
   NB(k) = ~ B(k);

}
```

Partea frumoasă este că în VHDL nici măcar acest ciclu nu este necesar. Limbajul **VHDL este un limbaj vectorial,** adică admite operanzi de tip vector. Dacă scriem:

```
NB <= not B;
```

înseamnă că se generează porţi **not** pentru fiecare element al vectorului **B**, de la stânga la dreapta. Se observă ca specificarea operaţiei nu depinde de lungimea vectorilor: 4 inversoare se vor specifica cu acelaşi efort ca 32 de inversoare!

Atenţie! `NB <= not B;` nu este o instrucţiune. Este o declaraţie! Efectul ei constă în generarea a 4 inversoare. Declaraţia `NB <= not B;` are acelaşi efect ca desenarea celor 4 inversoare.

Declaraţiile VHDL generează componente hardware !

Semnalul **NB** este un semnal intern. **Un semnal intern** este ieşirea unui bloc logic intern şi intrare altui bloc intern. Adică un semnal intern apare o singură dată în partea dreapta a unei declaraţii şi o dată sau de mai multe ori în partea stângă. Declaraţia de semnal intern are următoarea sintaxă:

```
signal nume_semnal    : tip_semnal;
```

Declaraţi semnalul NB iar apoi scrieţi `NB <= not B;` după **begin** şi salvaţi. Trebuie să obţineţi următorul cod:

```
architecture Behavioral of as4v is
   signal NB   : std_logic_vector(3 downto 0);
begin
   NB <= not B;

end Behavioral;
```

Apoi în fereastra **Sources** selectaţi acest fişier iar din fereastra **Processes** executaţi procesul **Check Syntax**. Dacă apar erori pe care nu le puteţi rezolva chemaţi profesorul!

Elemente ale limbajului VHDL (Nu săriţi această secţiune) – Operatorii logici. In VHDL există următorii operatori logici:

and or not nand nor xor xnor

Aceşti operatori acţionează atât asupra semnalelor scalare cât şi asupra celor vectoriale (magistrale). Operatorul **not**, fiind un operator unar are prioritatea cea mai mare. Pentru ceilalţi operatori **nu este definită prioritatea**. Din acest motiv este nevoie de paranteze! De exemplu, declaraţia:

```
Y<= A and B or C and D; --error
```

va genera o eroare.

Aceasta trebuie scrisă ca:

```
Y<= (A and B) or (C and D); --OK
```

Si declaraţia:

```
Y<= A nand B nand C; --error
```

este greşită trebuie rescrisă ca:

```
Y<= (A nand B) nand C;
```

sau ca

```
Y<= A nand (B nand C);
```

Se poate renunţa la paranteze dacă în expresia booleană intervin numai operatori de acelaşi fel şi în plus respectivul operator este asociativ (and, or, xor). De exemplu declaraţia:

```
Y<= A and B and C; --OK
```

este corectă.

În cazul în care expresia logică se foloseşte pentru asignarea unui vector, calculul se face **de la stânga la dreaptă**, element cu element, indiferent de tipul gamei şi de valorile indecşilor stânga şi dreapta. Evident toţi vectorii trebuie să aibă aceeaşi dimensiune. De exemplu:

```
    signal v1   :  std_logic_vector(1 to 4);
    signal v2   :  std_logic_vector(6 downto 3);
    signal rez  :  std_logic_vector(3 downto 0);
.........
rez  <= v1 and v2;
```

Se execută element cu element, de la stânga la dreapta astfel:

```
v1(1)      v1(2)      v1(3)      v1(4)
 and        and        and        and
v2(6)      v2(5)      v2(4)      v2(3)
  ↓          ↓          ↓          ↓
rez(3)     rez(2)     rez(1)     rez(0)
```

Pasul 4: Descrierea cele 4 MUX2.

În figura 5 este sunt prezentate detaliat cele 4 MUX2 şi aşa cum au fost definite în figura 1

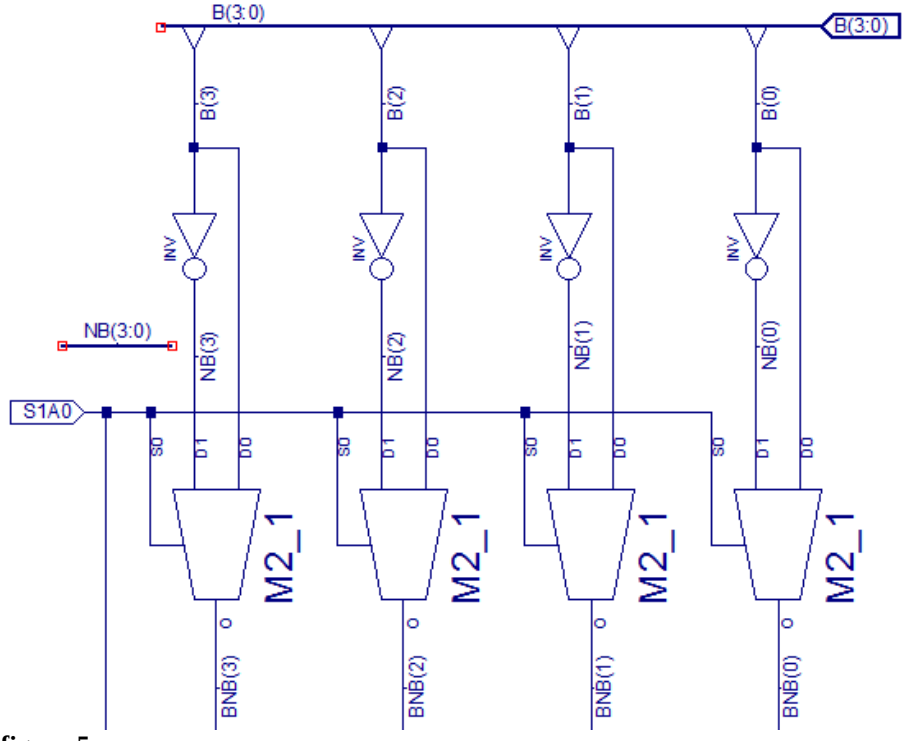


figura 5

Pentru a specifica cele patru MUX2 se va folosi **Conditional signal assignment** (nu ştiu cum se traduce). Această declaraţie este asemănătoare cu construcţia **if-elseif-elseif- ….else** din C. Spre deosebire de C, pe toate ramurile din partea stângă a asignărilor se află acelaşi obiect:

Sintaxa Conditional signal assignment:

```
target_signal <= expression1 when bcond1  else
                 expression2 when bcond2  else
                             ..........
                 expressionN when bcondN  else
                 expressionN+1;
```

- *Target_signal* poate fi scalar sau vector.
- *Target_signal* se calculează conform *expression1* dacă expresia booleană *bcond1* este adevărată. Altfel se evaluează următoarea condiţie, *bcond2*, şi dacă aceasta este adevărată *Target_signal* se calculează conform *expression2*. Dacă nici aceasta nu este adevărată se trece la *bcond3*, ş.a m.d.
- Dacă nici o condiţie nu este adevărată *Target_signal* se calculează conform *expressionN+1*.
- Condiţiile se pot suprapune.

Expresiile booleene de mai sus se construiesc după aceleaşi reguli ca în C. După cum s-a văzut în cazul operatorilor logici, notaţia operatorilor în VHDL diferă de notaţia folosită în C. Pentru a construi expresii care se evaluează la adevărat sau fals la fel ca în C sunt necesari operatorii logici. Pe lângă =, >, <, pentru care semnificaţie este evidentă, în VHDL mai avem /= pentru diferit, <= pentru mai mic sau egal şi >= pentru mai mare sau egal. La fel ca în C, **prioritatea operatorilor relaţionali este mai mare decât a celor logici**.

Compararea obiectelor scalare de tip **std_logic** este simplă: deoarece ‚1' este mai mare ca ‚0'. Pentru vectori regulile sunt mai complicate. Pentru a nu lungi prezentarea, în această fază ne vom mulţumi să comparăm **std_logic vectori** de **aceeaşi lungime**. Comparaţia se face de la stânga la dreaptă, element cu element, indiferent de gama şi de limitele vectorilor. Dacă va fi necesar se va reveni!

Pentru a genera cele 4 multiplexoare ar trebui scris:

```
BNB(0) <= B(0) when S1A0 ='0' else NB(0);
BNB(1) <= B(1) when S1A0 ='0' else NB(1);
BNB(2) <= B(2) when S1A0 ='0' else NB(2);
BNB(3) <= B(3) when S1A0 ='0' else NB(3);
```

Deoarece VHDL este un limbaj vectorial, în loc de cele 4 declaraţii de mai sus se poate scrie una singură:

```
BNB <= B when S1A0 ='0' else NB;
```

Declaraţi semnalul BNB şi scrieţi această declaraţie în interiorul arhitecturii, salvaţi şi apoi verificaţi sintaxa. Dacă apar erori pe care nu le puteţi rezolva chemaţi profesorul!

Pasul 5: Descrierea sumatorul pe 4 biţi.

În figura **6** este prezentat sumatorul pe 4 biţi din figura 1 aşa cum a fost desenat în laboratorul 3:

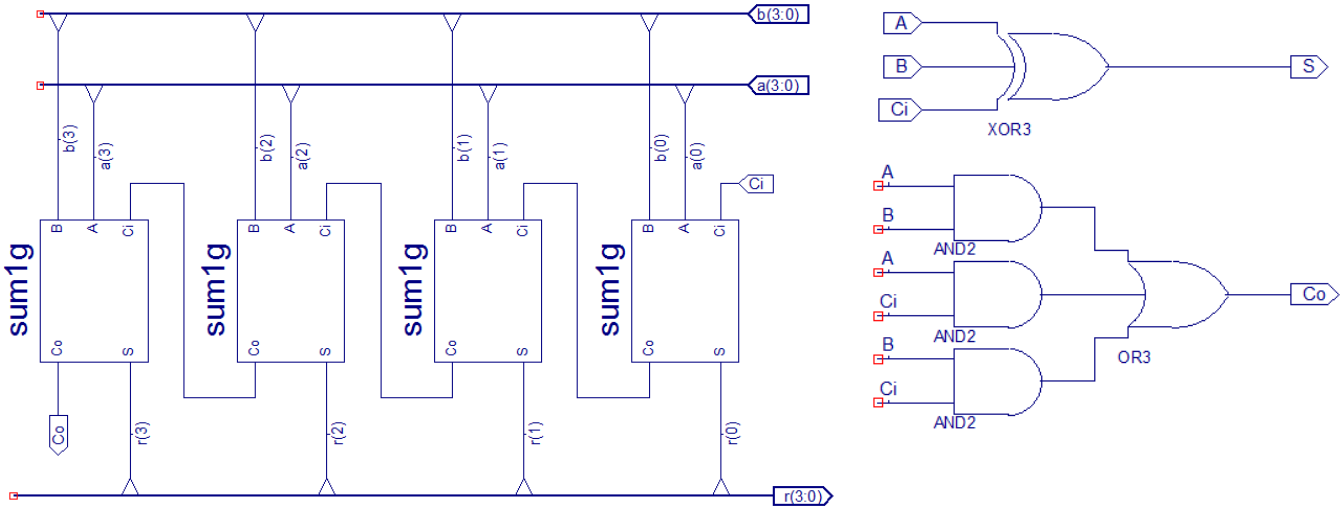


figura 6

Deoarece sumatorul pe patru biţi este alcătuit din patru sumatoare pe un bit (sum 1g) iar sumatorul de un bit se specifică prin intermediul a două asignări de semnal, pentru a genera cele 4 sumatoare **ar trebui** scris:

NU SCRIETI ACEST COD!

```
S(0)  <=   A(0) xor B(0)   xor Cin;
Cy(0) <= ( A(0) and B(0) ) or ( A(0) and Cin )   or (B(0) and Cin );

S(1)  <=   A(1) xor B(1)   xor Cy(0);
Cy(1) <= ( A(1) and B(1) ) or ( A(1) and Cy(0) ) or (B(1) and Cy(0) );

S(2)  <=   A(2) xor B(2)   xor Cy(1);
Cy(2) <= ( A(2) and B(2) ) or ( A(2) and Cy(1) ) or (B(2) and Cy(1) );

S(3)  <=   A(3) xor B(3) xor Cy(2);
```

Evident, acelaşi dezavantaj ca la not4. Dacă am trece la 32 de biţi ar trebui scrise 64 de declaraţii. Soluţia vectorială nu funcţionează în acest caz deoarece vectorul **Cy** nu se calculează din elemente cu acelaşi indice. Ar mai rămâne soluţia cu ciclu for. Această soluţie a fost aplicată pentru **supraîncărcarea** operatorului +, după cum urmează:

NU SCRIETI ACEST COD!

```
...
 carry := '0';
BV           := B;
for i in 0 to A'left loop
    sum(i)   := A(i) xor BV(i) xor carry;

    carry    := (A(i) and BV(i)) or (A(i) and carry) or (carry and BV(i));
end loop;
...
```

Supraîncărcarea este făcut într-un fişier care se include, la fel ca în C. În VHDL includerea se face cu directiva „**use**”:

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

Aceste fişiere sunt incluse automat atunci când se creează o nouă sursă de tip VHDL.

Supraîncărcarea este făcută în fişierul `STD_LOGIC_ARITH.` Funcţia care face supraîncărcarea consideră MSB bitul cel mai din stânga şi LSB bitul cel mai din dreapta, pentru ca adunarea să fie cât mai apropiată de modul uman de a aduna. Tipul gamei şi limitele vectorilor nu contează.

De asemenea, supraîncărcarea permite să adunăm şi vectori care nu au aceeaşi lungime. Mai mult, se permite adunarea unui vector cu un scalar de tip **std_logic** sau cu un semnal de tip **integer**. Dacă se adună doi vectori de lungimi diferite, cel mai scurt se extinde astfel încât ambii vectori să aibă aceeaşi lungime. Totuşi este obligatoriu ca vectorul din partea stângă a operatorului de asignare să aibă lungimea celui mai lung vector din partea dreaptă.

Supraîncărcarea operatorului ‚+’ permite să generăm hardware-ul necesar adunare scriind doar:

```
R  <= A + BNB + S1A0;
```

Scrieţi declaraţia de mai sus în interiorul arhitecturii, salvaţi şi apoi verificaţi sintaxa. Dacă apar erori pe care nu le puteţi rezolva chemaţi profesorul!

Pasul 6: Crearea schemei sumator/scăzătorului pe 4 biţi.

Se adaugă la proiect un nou fişier sursă de tip schematic ce va referi descrierea VHDL a sumator/scăzătorului pe 4 biţi prin intermediul unui simbol. Acest fişier este situat în vârful ierarhiei şi se va numi **top** sau **root** sau altfel, după preferinţă. Se creează **simbolul** pentru descrierea sumator/scăzătorului din fişierul **as4.vhd**. Pentru a crea un simbolul asociat unui fişier selectaţi respectivul fişier în fereastra **Sources** iar în fereastra **Processes** expandaţi **Design Utilities** şi lansaţi procesul **Create Schematic Symbol**. Apoi desenaţi schema din figura 7. Verificaţi schema şi faceţi implementarea.

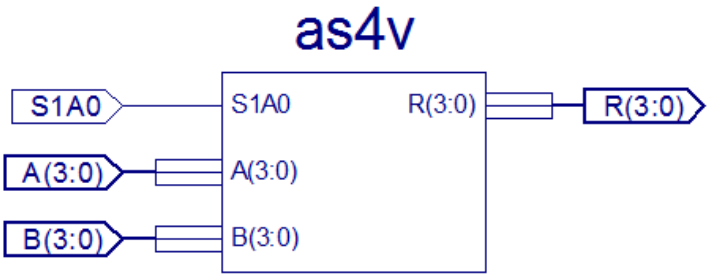


figura 7

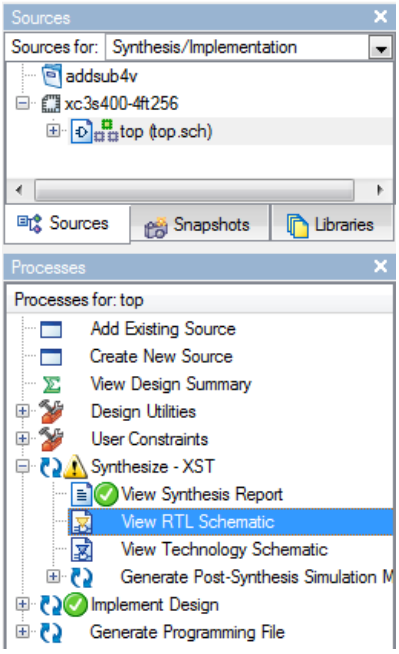


figura 8

Descrierile hardware VHDL sunt echivalente cu schemele. Acest fapt se poate verifica dacă se lansează procesul View RTL Schematic. În fereastra **Sources** selectaţi fişierul **top.sch** iar în fereastra **Processes** expandaţi **Synthesize**. În final lansaţi procesul **View RTL Schematic**, ca în **figura 8**. Va apare fereastra din **figura 9**. În figura se observă simbolul asociat lui top.sch:

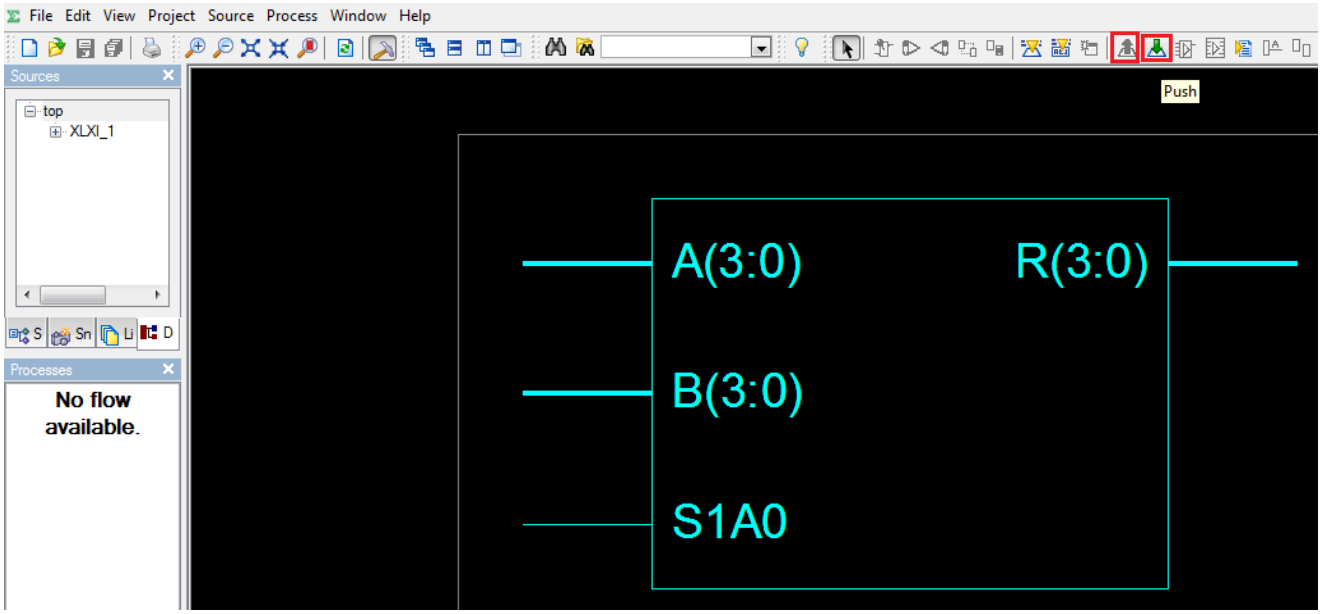


figura 9

Pentru a ajunge la schema generată pentru descriera VHDL a sumator/scăzătorului din fişierul as4v.vhd folosiţi butoanele de navigare în ierarhie **Push** şi **Pop** marcate cu roşu în **figura 9**.

Pentru a **vedea** ce este în interiorul unui simbol **selectaţi** respectivul simbol şi apăsaţi **Push**. După ce apăsaţi **Push** de două ori ar trebui să ajungeţi la o schemă foarte asemănătoare cu schema/sumator scăzătorului din laboratorul precedent. **La pasul următor vi se va cere să prezentaţi această schemă**.

Pasul 7: Simularea sumator/scăzătorului pe 4 biţi.

Operaţie (0→+, 1→ -)	A(3:0)	B(3:0)
+	+5	+2
+	-5	+2
+	+5	-2
+	-5	-2
-	+5	+2
-	-5	+2
-	+5	-2
-	-5	-2

Adăugaţi la proiect o nouă sursă de tip Test Bench Vaweform, cu numele tbw (sau altul, dacă acesta nu vă place), conform procedurii descrise în laboratorul 3 şi 4. În fereastra de dialog **Initial Timing…** ce apare în procesul de definire a stimulilor nu uitaţi să selectaţi opţiunea **Combinatorial** (dacă aceasta nu este deja selectată).

Procedaţi ca în laboratoarele precedente.

Verificarea se va face pentru 8 valori de intrare, conform tabelului alăturat.

Introduceţi aceste valori în editorul de stimuli şi faceţi simularea. Trebuie să obţineţi aceleaşi rezultate ca în laboratorul 4.

Dacă funcţionează corect şi în plus puteți prezenta schema RTL de la pasul 6, chemaţi profesorul pentru validare. Dacă aţi ajuns aici aveţi nota 5.

Pasul 8: Două întrebări (opţional).

Analizaţi schema din figura următoare:

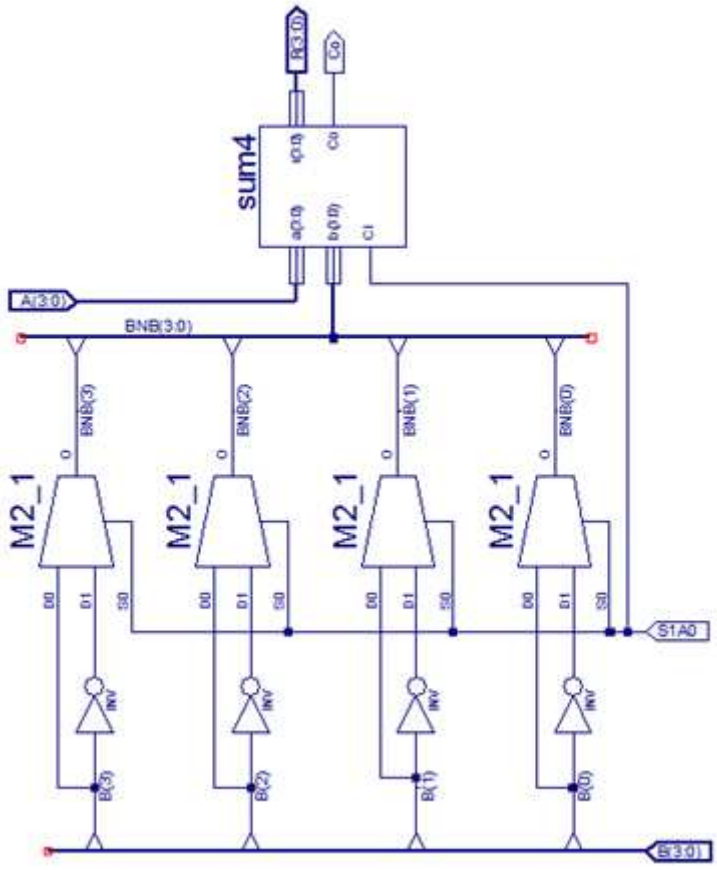


figura 10

Comparaţi-o cu schema din figura 1.

Prima întrebare: **Sunt cele două scheme echivalente din punct de vedere funcţional?**

Dacă parcurgem schema din figura 1 de sus în jos mai întâi întâlnim cele 4 inversoare care calculează inversul lui **B**. Descrierea în VHDL a celor 4 inversoare este:

```
NB   <= not B;
```

Sub inversoare se află cele 4 multiplexoare care generează la ieşire pe **B** sau pe not **B** în funcţie de S1A0. Descrierea în VHDL a celor 4 multiplexoare este:

```
BNB  <= B when S1A0 = '0' else NB;
```

În partea de jos a schemei este sumatorul a cărui descriere în VHDL este:

```
R    <= A + BNB + S1A0;
```

În fişierul as4v.vdh descrierea sumator/scăzătorului s-a făcut în ordinea prezentată mai sus.

Vom aplica aceeaşi procedură pentru schema din figura 10. Dacă parcurgem această schema tot de sus în jos rezultă următoarea descriere VHDL:

```
R     <= A + BNB + S1A0;
BNB   <= B when S1A0 ='0' else NB;
NB    <= not B;
```

A doua întrebare: Dacă inversăm ordinea declaraţiilor va mai funcţiona corect sumator/scăzătorul?

Descrieţi sumator/scăzătorul în ordine inversă, simulaţi şi aflaţi răspunsul. **Justificaţi!**

Veţi obţine între 0 şi 2 puncte. NU chemaţi profesorul.

Pasul 9: Doar două declaraţii VHDL.

Sumator/scăzătorul a fost descris prin intermediul a trei declaraţii. Descrierea se poate face doar cu două declaraţii. Încercaţi şi verificaţi că funcţionează corect prin simulare.

Dacă funcţionează aveţi între 0 şi 3 puncte.

Chemaţi profesorul pentru evaluarea la punctele 8 şi 9.