

Laboratorul 2 - VHDL

1. Introducere

Limbajul de modelare hardware VHDL (*Very High Speed Integrated Circuit Hardware Description Language*) suportă trei tipuri distincte de metode de modelare (descriere) a circuitelor:

1. a fluxului de date,
2. structurală și
3. comportamentală.

În plus, față de modelările prezentate anterior, VHDL-ul acceptă și o modelare mixtă – deci o combinație între oricare două sau multe din modelările anterioare.

Modelările prin metodele fluxului de date și structurală sunt utilizate în modelarea circuitelor combinaționale. Modelarea comportamentală este utilizată atât pentru circuitele combinaționale cât și pentru cele secvențiale.

Cel mai înalt nivel de abstractizare este nivelul de descriere al comportării (funcționării) numit în engleză **behavioral**. La acest nivel de abstractizare un sistem este descris prin ceea ce face, adică prin modul cum se comportă și nu prin componentele sale și conexiunile dintre acestea. O descriere de acest tip specifică relațiile dintre semnalele de intrare și ieșire.

Structura limbajului VHDL

Un modul VHDL are o structură foarte bine definită care îi permite să fie structurat într-o manieră clară și logică. Un modul VHDL are 2 mari componente:

1. o entitate (**entity**) principală și
2. un corp arhitectural.

Deci, fiecare modul este caracterizat de o declarație a entității și de corpul arhitectural. **Declarația entității se poate considera o interfață cu mediul extern** care **definește semnalele de intrare și de ieșire**, pe când **corpul arhitectural conține descrierea entității** și este compus din alte entități, procese și componente interconectate.

```
entity example_code is
port (
    port_1 : in  std_logic;
    port_2 : out std_logic_vector(1 downto 0)
);
end example_code
```

```

architecture example_code_arch of example_code is
...
begin
...
end example_code_arch;

```

Entitatea (**entity**) poate conține numele porturilor, dimensiunea porturilor și direcția (intrare sau ieșire). Forma generală de declarare a unei entități este următoarea:

```

entity NUME_ENTITATE is [ generic (declarații_generice);]
    port (  nume_semnale : mode tip ;
            nume_semnale : mode tip ;
            :
            nume_semnale : mode tip;
    end [NUME_ENTITATE];

```

O entitate începe întotdeauna cu cuvântul cheie **entity** (entitate) urmat de numele entității și de cuvântul cheie **is** (este). În continuare sunt declarațiile porturilor cu cuvântul cheie **port**. Declarația entității se termină întotdeauna cu cuvântul cheie **end**. NUME_ENTITATE este un identificator (simbol) ales de utilizator.

Componente:

1. **nume_semnale** - constă într-o listă de identificatori separați prin punct virgulă care specifică semnalele interfeței externe;
2. **mode** este un cuvânt rezervat care indică sensul semnalelor. Astfel avem:
 - **in** - semnal de intrare;
 - **out** - semnal de ieșire, valoarea poate fi citită doar de alte entități;
 - **buffer** - semnal este de ieșire, dar valoarea poate fi citită în arhitectura entității;
 - **inout** - semnal ce poate fi ieșire sau intrare (bidirecțional).
3. **tip** este tipul de semnal. Potențiale exemple de tipuri de semnal sunt: **bit**, **bit_vector**, **boolean**, **character**, **std_logic**, **std_ulogic**, **std_logic_vector**, etc.

Corpul arhitectural specifică modul în care funcționează și în care este implementată o entitate. Corpul arhitectural poate fi compus din următoarele 3 secțiuni de bază:

1. declarații de componente,
2. declarații de semnale și
3. zona cod de descriere funcțională.

De exemplu, mai jos observăm zona declarațiilor componentelor (utilizate în proiectarea ierarhică a sistemului) și zona declarațiilor semnalelor – acestea sunt utilizate în realizarea conexiunilor locale între diferitele blocuri în zona codului ce descrie funcțional circuitul.

```

...
architecture example_code_arch of NUME_ENTITATE is
    component instanță1 port (
        port_in : in std_logic;
    ...

```

```

        port_out : out std_logic
    );

    component instance2 port (
        ...
        port_out4 : out std_logic_vector(3 downto 0);
    );

    signal sig_a : std_logic;
    signal sig_b : std_logic_vector(1 downto 0);
    signal sig_c : std_logic_vector(3 downto 0);
    ...
begin
    comp1 : instance1
    port map (
        port_in => sig_a;
        ...
        port_out => sig_b;
    );

    comp2 : instance2
    port map (
        ...
        port_out4 => sig_c;
    );

    comp3 : instance1
    port map (
        ...
    );

    sig_a <= port_1 and sig_c(0);

    process (sig_c) begin
        port_2 <= sig_c(3 downto 1);
    end process;

end example_code_arch;

```

Structura cea mai generală a corpul arhitectural arată în felul următor:

```

architecture nume_arhitectura of NUME_ENTITATE is
    -- Declaratii
        -- declarații ale componentelor
        -- declarații de semnale
        -- declarații de constante

```

```

-- declarații de funcții
-- declarații de proceduri
-- declarații de tipuri

begin
    -- Instrucțiuni
end nume_arhitectura;
```

2. Modelarea de tip flux de date

Modelarea de tip flux de date are la bază operația de asignare în care o valoare este asignată la un anumit semnal. Sintaxa operației de asignare este următoarea:

Left_signal <= Right_expression;

Left_signal este destinația și poate fi pe unul sau mai mulți biți. **Right_expression** este o expresie ce poate utiliza unul sau mai mulți operatori. În cele de mai jos se prezintă o modelare de tip flux de date a unui circuit ce are o ieșire și două intrări.

```

entity AND_gate is
port ( a : in  std_logic;
       b : in  std_logic;
       c : out std_logic
);
end AND_gate;

architecture AND_gate_dataflow_arch of AND_gate is

begin
    c <= a and b;
end AND_gate_dataflow_arch;
```

Mai jos sunt prezentate câteva exemple de diferite tipuri de asignări:

```

z <= x or y;
z <= v and w and x and y;
z <= v and w or x nor y;
```

Semnalele sunt actualizate imediat când este executat instrucțiunea de asignare (ca mai sus) sau cu o anumită întârziere (după cum se arat mai jos):

```

z <= (a xor b) after 2 ns;
```

În particular se poate astfel specifica și o formă de undă:

```

signal unda : std_logic;
...
unda <= '0', '1' after 5ns, '0' after 10ns, '1' after 20ns;
```

Diferențele dintre variabile și semnale sunt importante și constau în special în momentul la care acestea își schimb valoarea. O variabilă este actualizată imediat când este executată instrucțiunea de actualizare. Semnalele își modific valoarea cu o anumită întârziere după ce a fost evaluat expresia de asignare.

Interconexiunea dintre diferite obiecte trebuie realizată prin semnale (fire). Aceste “fire” (semnale) pot fi scalare sau vectoriale și trebuie în mod obligatoriu definite înainte de a fi utilizate. De exemplu:

```
signal y      : std_logic;           -- fir scalar
signal sum    : std_logic_vector(3 downto 0); -- semnal vectorial
```

În exemplul anterior, **STD_LOGIC** definește un semnal scalar (un bit), în timp ce **STD_LOGIC_VECTOR** definește un “semnal” ce are o anumită lățime în biți (4 în situația specifică prezentată anterior).

Există un număr foarte mare de operatori în VHDL. **Pentru o descriere mai amănunțită a lor va rog să consultați Anexa 1.** De exemplu, operatorul **&** concatenează două semnale. De exemplu:

```
signal m      : STD_LOGIC;           -- date scalare
signal switches : STD_LOGIC_VECTOR (7 downto 0); -- date vectoriale
...
switches <= switches(7 downto 1) & m;
```

După cum se vede mai sus operatorul de concatenare, concatenează semnalul **m** (drept ultim bit) cu primii 7 biți (cei mai semnificativi) ai semnalului **switches** formând în final noul semnal **switches**.

Pentru intrări ce necesită valori fixe (realizabile prin intermediul unor rezistoare de tip *pull-up* și/sau *pull-down*) în VHDL se utilizează sintaxa ‘H’ sau ‘L’ (deci valori constante ce sunt întotdeauna de tip **STD_LOGIC**). De exemplu:

```
z <= 'H';
y <= 'L';
```

Implementarea unui multiplexor 2:1

Implementați în VHDL un multiplexor 2:1 (vezi **Figura 1**) prin intermediul unei modelări de tip flux de date. Testați funcționarea corectă a acestui multiplexor cu autorul plăcii **Basys 3** de la Digilent.

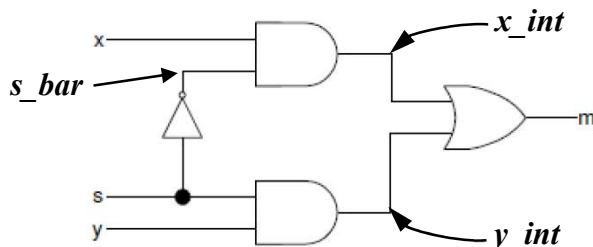


Figura 1. Schema hardware a unui multiplexor cu 2 intrări

Descrierea implementării în VHDL este prezentată mai jos:

```

entity MUX_CI is
port (
    x : in  STD_LOGIC;
    y : in  STD_LOGIC;
    s : in  STD_LOGIC;
    m : out STD_LOGIC
);
end MUX_CI;

architecture behavior of MUX_CI is

    Signal s_bar : STD_LOGIC;
    Signal x_int : STD_LOGIC;
    Signal y_int : STD_LOGIC;
begin
    s_bar <= not s;
    x_int <= x and s_bar;
    y_int <= y and s;
    m <= x_int or y_int;
end behavior;

```

Implementarea a 2 multiplexoare 2:1 cu pin comun de selecție

Implementați în VHDL două multiplexoare 2:1 (vezi **Figura 1**) ce au același pin de selecție prin intermediul unei modelări de tip flux de date. Testați funcționarea corectă a acestui multiplexor cu autorul plăcii **Basys 3** de la Digilent.

Descrierea implementării în VHDL este prezentată mai jos:

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

library UNISIM;
use UNISIM.VComponents.all;

Entity mux_2x2_to_1 is
port (
    x : in  STD_LOGIC_VECTOR(1 downto 0);
    y : in  STD_LOGIC_VECTOR(1 downto 0);
    s : in  STD_LOGIC;
    m : out STD_LOGIC_VECTOR(1 downto 0)
);
end mux_2x2_to_1;

Architecture behavior of mux_2x2_to_1 is

```

```

Signal s_bar : STD_LOGIC;
Signal x_int : STD_LOGIC_VECTOR(1 downto 0);
Signal y_int : STD_LOGIC_VECTOR(1 downto 0);

begin

    s_bar <= not s;

    x_int(1) <= x(1) and s_bar;
    y_int(1) <= y(1) and s;
    m(1) <= x_int(1) or y_int(1);

    x_int(0) <= x(0) and s_bar;
    y_int(0) <= y(0) and s;
    m(0) <= x_int(0) or y_int(0);

end behavior;

```

Implementarea unui multiplexor 2:1 cu ajutorul instrucțiunilor de asignare condiționată

Instrucțiunea de asignare condițională (ce va fi studiată în cele ce urmează) este echivalentă funcțional cu instrucțiunea condițională **if** și are sintaxa următoare:

```

semnal <= [expresie when condiție else ...]
           expresie;

```

Valoarea uneia din expresiile sursă se atribuie semnalului destinație. Expresia atribuită va fi prima a cărei condiție booleană asociată este adevărată. La execuția unei instrucțiuni de asignare condițională, condițiile sunt testate în ordinea în care ele sunt scrise. La întâlnirea primei condiții care se evaluează la valoarea booleană **TRUE**, expresia corespunzătoare acesteia se asignează semnalului destinație. Dacă nici o condiție nu se evaluează la valoarea **TRUE**, semnalului destinație i se asignează ultima expresie, cea a ultimei clauze **else**. Dacă există două sau mai multe condiții care se evaluează la valoarea **TRUE**, va fi luată în considerare doar prima dintre acestea.

Deosebirile dintre instrucțiunea de asignare condițională și instrucțiunea condițională **if** sunt următoarele:

1. Instrucțiunea de asignare condițională **este o instrucțiune concurentă**, deci poate fi utilizată într-o arhitectură, în timp ce instrucțiunea **if** este secvențială, astfel că poate fi utilizată **numai** în interiorul unui proces.
2. Instrucțiunea de asignare condițională poate fi utilizată numai pentru asignarea valorii unor semnale, în timp ce instrucțiunea **if** poate fi utilizată pentru execuția oricărei instrucțiuni secvențiale.

În exemplul următor se definește o entitate și două arhitecturi pentru o poartă SAU EXCLUSIV cu două intrări. Prima arhitectură utilizează o instrucțiune de asignare condițională, iar a doua utilizează o instrucțiune **if** echivalentă.

```

entity xor2 is
    port (
        a, b : in  STD_LOGIC;
        x   : out STD_LOGIC
    );
end xor2;

architecture arh1_xor2 of xor2 is
begin
    x <= '0' when a = b else
        '1';
end arh1_xor2;

architecture arh2_xor2 of xor2 is
begin
    process (a, b)
    begin
        if a = b then x <= '0';
        else x <= '1';
        end if;
    end process;
end arh2_xor2;

```

Exercițiu 2.1: descrieți printr-o modelare de tip flux de date funcționarea unui multiplexor 2:1 utilizând pentru aceasta asignarea condiționată (deci, utilizați instrucțiunile **when** și **else**).

3. Modelarea structurală

Modelarea structurală impune o proiectare ierarhizată în care se definesc diferite componente. Aceste componente sunt utilizate/reutilizate ulterior de mai multe ori. Componentele sunt elementele de bază ce definesc funcționalități elementare sau mai complexe a unui circuit. Odată ce au fost definite, aceste componente pot fi folosite ca blocuri structurale într-o entitate la un nivel mai înalt. În acest mod se reduce în mod semnificativ complexitatea proiectelor mari. Componentele pot fi alte module sau/și diferite primitive

La fel ca în orice aplicație software proiectele ierarhizate sunt preferate întotdeauna în locul celor pe un singur nivel.

```

entity AND_gate_structural is
port ( a: in  std_logic;
       b: in  std_logic;
       c: in  std_logic;
       d: out std_logic
);

```



```
architecture AND_gate_struct of AND_gate_structural is
```

```
    component and2 port  
    (  
        i0, i1    : in    std_logic;  
        O        : out  std_logic  
    ) end component;
```

```
    Signal a_int : STD_LOGIC;
```

```
begin
```

```
    and_comp_1 : and2 port map (  
        i0 => a,  
        i1 => b,  
        o => a_int  
    );
```

```
    and_comp_2 : and2 port map (  
        i0 => a_int,  
        i1 => c,  
        o => d  
    );
```

```
end AND_gate_arch;
```

După cum s-a observat și în exemplul anterior, componentele pot fi conectate între ele prin intermediul **semnalelor (Signal)** declarate în secțiunea **architecture**. Un alt exemplu, în care se utilizează 2 componente și un semnal este prezentat mai jos.

```
entity AND_OR_gate_structural is  
port (a : in    std_logic;  
      b : in    std_logic;  
      c : in    std_logic;  
      d : out  std_logic;  
);
```

```
architecture AND_OR_gate_struct of AND_OR_gate_structural is
```

```
    component and2 port  
    (  
        i0, i1 : in    std_logic;  
        o      : out  std_logic  
    ) end component;
```

```
    component or2 port  
    (  
        i0, i1 : in    std_logic;  
        o      : out  std_logic  
    ) end component;
```

```
    Signal e : std_logic;
```

```

begin
    and_comp : and2 port map (
        i0 => a;
        i1 => b;
        o  => e;
    );

    or_comp : or2 port map (
        i0 => c;
        i1 => e;
        o  => d;
    );

end AND_gate_arch;

```

Exercițiu 2.2: pentru cele 2 circuite descrise anterior prin intermediul unei modelări structurale desenați schema echivalentă rezultată.

Două multiplexoare de tip 2:1 cu pin comun de selecție

Creați un modul VHDL ce va avea 4 intrări $x0$, $x1$, $y0$ și $y1$ (x și y sunt intrările unui multiplexor 2:1) o intrare de selecție (s – unică celor 2 multiplexoare) și 2 ieșiri $m0$, $m1$ utilizând o modelare structurală. Codul implementării în VHDL se prezintă mai jos.

```

entity mux_2x2_to_1_structural is
    port (
        x : in  STD_LOGIC_VECTOR(1 downto 0);
        y : in  STD_LOGIC_VECTOR(1 downto 0);
        s : in  STD_LOGIC;
        m : out STD_LOGIC_VECTOR(1 downto 0)
    );
end mux_2x2_to_1_structural;

architecture behavior of mux_2x2_to_1_structural is

    component and2 port
        ( i0, i1: in  std_logic;
          o  : out std_logic);
    end component;

    component or2 port
        ( i0, i1: in  std_logic;
          o  : out std_logic);
    end component;

    component inv port
        ( i  : in  std_logic;
          o  : out std_logic);
    end component;

    signal s_bar : std_logic;
    signal x_int : std_logic_vector(1 downto 0);
    signal y_int : std_logic_vector(1 downto 0);

```

```

begin
    n1 : inv port map ( i => s,          o => s_bar);

    a1 : and2 port map ( i0 => x(1),    i1 => s_bar,    o => x_int(1));
    a2 : and2 port map ( i0 => y(1),    i1 => s,        o => y_int(1));
    o1 : or2  port map ( i0 => x_int(1), i1 => y_int(1), o => m(1));

    a3 : and2 port map ( i0 => x(0),    i1 => s_bar,    o => x_int(0));
    a4 : and2 port map ( i0 => y(0),    i1 => s,        o => y_int(0));
    o2 : or2  port map ( i0 => x_int(0), i1 => y_int(0), o => m(0));

end behavior;

```

4. Modelarea comportamentală

Modelarea comportamentală (*behavioral*) este utilizată în descrierea circuitelor complexe. În VHDL, modelarea comportamentală este realizată în cadrul corpului arhitecturii – aici procesele (**processes**) sunt definite ca circuite secvențiale. Mecanismul de modelare comportamentală se prezintă mai jos (într-o descriere generică):

```

-- următorul proces este utilizat doar la inițializarea semnalelor
-- la începutul rulării
process begin
    a <= '1';
    b <= '0';
    ...
    wait;
end process;

-- următorul proces rulează de fiecare dată când orice semnal
-- din lista argumentelor își schimbă starea
process (c, d, e)
begin
    <behavioral code here>
end process;

```

Un modul poate conține un număr arbitrar de declarații de procese care la rândul lor pot conține alte declarații interne lor. Acestea interne poartă numele de declarații procedurale. **Toate procesele sunt executate în mod concurențial.** Deci, ordinea în care ele apar în model nu contează. În schimb declarațiile procedurale se execută în mod secvențial în ordinea în care ele au fost definite.

Primul proces, din exemplul anterior se execută în primul moment (la timpul $t = 0$). O declarație de tip **wait** este necesară a fi introdusă la sfârșit pentru a opri acest proces să se execute în mod continuu încă o dată și încă o dată. Procesele cu o listă de argumente sunt executate continuu, în restul timpului, de fiecare dată când unul din argumente își schimbă starea.

Mai jos se prezintă structura fundamentală de descriere a unui multiplexor într-o abordare comportamentală.

```

signal m : STD_LOGIC_VECTOR;

```

```

...
process (x, y, s)
begin
    if (s='0') then
        m <= y;
    else
        m <= x;
    end if;
end;

```

Multiplexor 2:1 descris printr-o modelare comportamentală

Implementați și testați funcționarea corectă a multiplexorului:

```

entity MUX2_1_behavioral is
Port (
    x : in STD_LOGIC;
    y : in STD_LOGIC;
    s : in STD_LOGIC;
    m : out STD_LOGIC
);
end MUX2_1_behavioral;

architecture Behavioral of MUX2_1_behavioral is

    Signal m_int : STD_LOGIC;

begin

    m <= m_int;

    process (x, y, s)
    begin
        if (s = '0') then
            m_int <= y;
        else
            m_int <= x;
        end if;
    end process;

end Behavioral;

```

Două multiplexoare de tip 2:1 cu pin comun de selecție

Implementați și testați funcționarea corectă a două multiplexoare ce au aceeași selecție. Implementarea este de tip comportamental.

Instrucțiunea **if** selectează pentru execuție una sau mai multe secvențe de instrucțiuni, în funcție de valoarea unei condiții corespunzătoare secvenței respective. Sintaxa acestei instrucțiuni este următoarea:

```

if condiție then secvență_de_instrucțiuni
    [elsif condiție then secvență_de_instrucțiuni...]
    [else secvență_de_instrucțiuni]
end if;

```

Fiecare condiție este o expresie booleană, care se evaluează la valoarea **TRUE** sau **FALSE**. Pot exista mai multe clauze **elsif**, dar poate exista o singură clauză **else**. Se evaluează mai întâi condiția de după cuvântul cheie **if**, și dacă este adevărată, se execută secvența de instrucțiuni corespunzătoare. În caz contrar, dacă este prezentă clauza **elsif**, se evaluează condiția de după această clauză și dacă această condiție este adevărată, se execută secvența de instrucțiuni corespunzătoare. În caz contrar, dacă sunt prezente alte clauze **elsif**, se continuă cu evaluarea condiției acestor clauze. Dacă nici o condiție evaluată nu este adevărată, se execută secvența de instrucțiuni corespunzătoare clauzei **else**, dacă aceasta este prezentă.

```

entity MUX2_1x2 is
Port (
    x : in  STD_LOGIC_VECTOR (1 downto 0);
    y : in  STD_LOGIC_VECTOR (1 downto 0);
    s : in  STD_LOGIC;
    m : out STD_LOGIC_VECTOR (1 downto 0)
);
end MUX2_1x2;

architecture Behavioral of MUX2_1x2 is

    Signal m_int : STD_LOGIC_VECTOR(1 downto 0);

begin
    m <= m_int;

    process (x, y, s)
    begin
        if (s = '0') then
            m_int <= y;
        else
            m_int <= x;
        end if;
    end process;

end Behavioral;

```

Descrierea unui multiplexor de tip 2:1 cu ajutorul instrucțiunii **case**

Ca și instrucțiunea **if**, instrucțiunea **case** selectează pentru execuție una din mai multe secvențe alternative de instrucțiuni pe baza valorii unei expresii. Spre deosebire de instrucțiunea **if**, în cazul instrucțiunii **case** expresia nu trebuie să fie booleană, ci poate fi reprezentată de un semnal, o variabilă sau o expresie de orice tip discret (un tip enumerat sau întreg) sau un tablou unidimensional de caractere (inclusiv **bit_vector** sau **std_logic_vector**). Instrucțiunea **case** se utilizează atunci când există un număr mare de alternative posibile. Această instrucțiune este mai lizibilă decât o instrucțiune **if** cu un număr mare de ramuri, permițând identificarea ușoară a unei valori și a secvenței de instrucțiuni asociate.

Sintaxa instrucțiunii **case** este următoarea:

```
case expresie is
    when opțiuni_1 => secvență_de_instrucțiuni
    ...
    when opțiuni_n => secvență_de_instrucțiuni
    [when others    => secvență_de_instrucțiuni]
end case;
```

Instrucțiunea **case** conține mai multe clauze **when**, fiecare specificând una sau mai multe opțiuni. Opțiunile reprezintă fie o valoare individuală, fie un set de valori cu care se compară expresia instrucțiunii **case**. În cazul în care expresia este egală cu valoarea individuală sau cu una din valorile setului, se execută secvența de instrucțiuni specificată după simbolul **=>**. O secvență de instrucțiuni poate fi formată și din instrucțiunea nulă, **null**. Spre deosebire de anumite limbaje de programare, instrucțiunile dintr-o secvență nu trebuie incluse între cuvintele cheie **begin** și **end**. Clauza **others** se poate utiliza pentru a specifica execuția unei secvențe de instrucțiuni în cazul în care valoarea expresiei nu este egală cu nici una din valorile specificate în clauzele **when**.

În cazul în care o opțiune este reprezentată de un set de valori, se pot specifica fie valorile individuale din set, separate prin simbolul “|” având semnificația “sau”, fie domeniul valorilor, fie o combinație a acestora, după cum se arată în exemplul următor.

```
case expresie is
    when val =>
        secvență_de_instrucțiuni
    when val1 | val2 | ... | valn =>
        secvență_de_instrucțiuni
    when val3 to val4 =>
        secvență_de_instrucțiuni
    when val5 to val6 | val7 to val8 =>
        secvență_de_instrucțiuni
    ...
    when others =>
        secvență_de_instrucțiuni
end case;
```

Exercițiu 2.3: descrieți comportamental funcționarea unui multiplexor 2:1 utilizând pentru aceasta instrucțiunea de selecție **case**.

5. Modelare mixtă

Sistemele complexe pot fi descrise în VHDL printr-o modelare mixtă. Această modelare are avantajul de a permite o descriere ierarhică a circuitului.

Un exemplu clasic de modelare mixtă este dat de implementarea unui multiplexor 3:1 prin utilizarea a două instanțe a unui multiplexor de tipul 2:1.

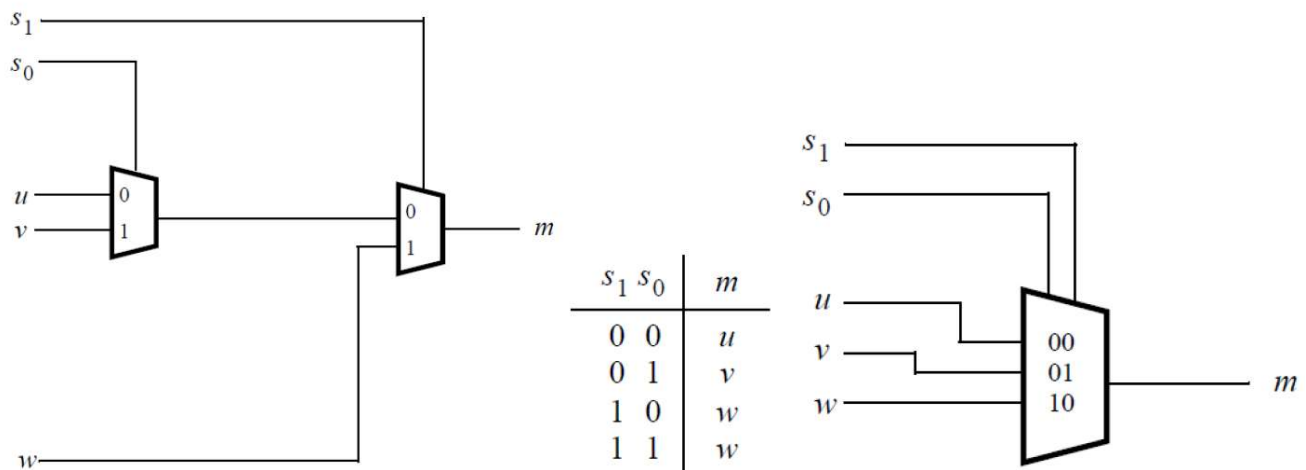


Figura 2. Modalitate de implementare a unui multiplexor 3:1

În implementarea din **Figura 2** u , v și w sunt intrările, S_0 și S_1 sunt pinii de selecție în timp ce ieșirea este m .

Pentru realizarea acestui multiplexor, codul modulului superior (**top module**) este cel prezentat mai jos. Pentru cele două instanțe de tip multiplexor 2:1 utilizați codul multiplexorului 2:1 dezvoltat în cadrul modelării structurale.

```
entity MUX3to1 is
  Port (
    u : in  STD_LOGIC_VECTOR(1 downto 0);
    v : in  STD_LOGIC_VECTOR(1 downto 0);
    w : in  STD_LOGIC_VECTOR(1 downto 0);
    S0 : in  STD_LOGIC;
    S1 : in  STD_LOGIC;
    m : out STD_LOGIC_VECTOR(1 downto 0));
end MUX3to1;

architecture Behavioral of MUX3to1 is
```

```

Component mux_2x2_to_1_structural
port (
    x  : in  STD_LOGIC_VECTOR (1 downto 0);
    y  : in  STD_LOGIC_VECTOR (1 downto 0);
    s  : in  STD_LOGIC;
    m  : out STD_LOGIC_VECTOR(1 downto 0)
);
end component;

Signal u1_o : STD_LOGIC_VECTOR(1 downto 0);
Signal m_int : STD_LOGIC_VECTOR(1 downto 0);

begin
    U1: mux_2bit_2_to_1_structural PORT MAP (
        x => u,
        y => v,
        s => s0,
        m => u1_o
    );


    U2: mux_2bit_2_to_1_structural PORT MAP (
        x => u1_o,
        y => w,
        s => s1,
        m => m_int
    );

    m <= m_int;

end Behavioral;

```

Dacă aveți dificultăți în selectarea unui modul pentru a deveni Top Module urmați pașii:

1. în tab-ul **Sources**;
2. Selectați fișierul sursă ce conține modulul pe care vreți să-l definiți de tip principal (*top module*);
3. Dați **Click dreapta** -> și alegeți **Set as Top Module**.
4. În acest moment codul sursă selectat devine **top module**. Acest lucru va fi indicat prin icoana  ce precede numele modulului. Acest modul devine acum rădăcina ierarhiei create.

Anexa 1. Operatori în VHDL

Limbajul VHDL suport diferite clase de operatori care operează pe semnale, variabile și constante. Clasele de operatori sunt prezentate în **Tabelul 1**.

Tabelul 1. Clasele de operatori

Nr.	Clasa	Operatori					
		and	or	nand	nor	xor	xnor
1.	Operatori logici						
2.	Operatori relaționali	=	/=	<	<=	>	>=
3.	Operatori de deplasare	sll	srl	sla	sra	rol	ror
4.	Operatori aditivi	+	=	&			
5.	Operatori unari	+	-				
6.	Operatori multiplicativi	*	/	mod	rem		
7.	Alți operatori	**	abs	not			

Prioritatea operatorilor este maximă pentru operatorii din clasa 7 și minimă pentru operatorii din clasa 1, prioritatea scăzând proporțional cu rangul operatorului. Dacă nu sunt folosite paranteze, operatorii din clasa 7 sunt aplicați prima dată, după care urmează celelalte clase în ordinea priorității. Operatorii din aceeași clasă au aceeași prioritate și sunt aplicați de la dreapta la stânga într-o expresie.

Operatorii logici (and, or, nand, nor, xor și xnor) sunt definiți pentru tipurile de date **“bit”**, **“boolean”**, **“std_logic”**, **“std_ulogic”** și pentru vectorii ai acestor tipuri de date. Aceștia sunt folosiți pentru a defini expresii logice booleene sau pentru a face operații la nivel de bit între șiruri de biți. Rezultatul acestor operatori este de tipul operanzilor implicați. Acești operatori pot fi aplicați și semnalelor, variabilelor sau constantelor. Trebuie reținut că operatorii nand și nor nu sunt asociativi și trebuie folosite paranteze într-o secvență de nand și nor pentru a preveni erori de sintaxă. De exemplu:

$$Z \leq X \text{ nand } Y \text{ nand } Z$$

va da o eroare de sintaxă și va trebui scris:

$$Z \leq (X \text{ nand } Y) \text{ nand } Z.$$