Compte rendu TPs systèmes embarqués

TP1: Modélisation d'un circuit combinatoir simple

Le but de ce premier TP est de réaliser un additionneur de deux nombres composés de 16 bits. Cet additionneur sera composé de 4 additionneurs 4 bits liés, qui à leurs tours seront composés de 4 additionneurs 1 Bits.

Par conséquent, on doit commencer tout d'abord par la réalisation d'un additionneur 1 Bits.

1 - Additionneur 1 Bits

Nous allons tout d'abord exposer le code de notre additionneur, expliquant par la suite chaque partie et le fonctionnement de notre additionneur.

Additionneur1:

```
----------------------------------------------------------------------------------
-
-- Company:
-- Engineer:
--
-- Create Date:    12:25:14 11/13/2023
-- Design Name:
-- Module Name:    additionneur1 - Behavioral
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
-
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
```

```
-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity additionneur1 is
generic(
    timeS: time := 2 ns;
    timeCout: time := 2 ns
);
    Port ( A : in  STD_LOGIC;
           B : in  STD_LOGIC;
           CIN : in  STD_LOGIC;
           S : out  STD_LOGIC;
           COUT : out  STD_LOGIC);
end additionneur1;

architecture Behavioral of additionneur1 is

begin
    S<=A xor B xor CIN after timeS ;
    COUT<= (A and B) or (B and CIN) or (A and CIN) after timeCout ;

end Behavioral;
```

Additionneur1_test

```
--------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:   12:27:04 11/13/2023
-- Design Name:
-- Module Name:   /home/ise/PartageVM/Partage/TP1_WM/additionneur1_test.vhd
-- Project Name:  TP1
-- Target Device:
-- Tool versions:
-- Description:
--
-- VHDL Test Bench Created by ISE for module: additionneur1
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
-- Notes:
-- This testbench has been automatically generated using types std_logic and
-- std_logic_vector for the ports of the unit under test.  Xilinx recommends
-- that these types always be used for the top-level I/O of a design in order
```

```
-- to guarantee that the testbench will bind correctly to the post-implementation
-- simulation model.
--------------------------------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--USE ieee.numeric_std.ALL;

ENTITY additionneur1_test IS
END additionneur1_test;

ARCHITECTURE behavior OF additionneur1_test IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT additionneur1
    PORT(
         A : IN  std_logic;
         B : IN  std_logic;
         CIN : IN  std_logic;
         S : OUT  std_logic;
         COUT : OUT  std_logic
        );
    END COMPONENT;


   --Inputs
   signal A : std_logic := '0';
   signal B : std_logic := '0';
   signal CIN : std_logic := '0';

    --Outputs
   signal S : std_logic;
   signal COUT : std_logic;
   -- No clocks detected in port list. Replace <clock> below with
   -- appropriate port name


BEGIN

    -- Instantiate the Unit Under Test (UUT)
   uut: additionneur1 PORT MAP (
          A => A,
          B => B,
          CIN => CIN,
          S => S,
          COUT => COUT
        );


    A<='0' after 0 ns, '1' after 10 ns,  '0' after 20 ns,  '1' after 30 ns,  '0'
after 40 ns,  '1' after 50 ns,  '0' after 60 ns,  '1' after 70 ns,  '0' after 80
ns;
```

```
    B<='0' after 0 ns, '1' after 20 ns,  '0' after 40 ns,  '1' after 60 ns,  '0'
after 80 ns;
    CIN<='0' after 0 ns,  '1' after 40 ns,  '0' after 80 ns;


END;
```

- Commençons alors par une exposition du code de l'additionneur 1.

Entité additionneur1:

L'additionneur1bits est composé de 3 ports d'entrée:

A ,B : Représentent les deux bits qu'on veut additionner.

Cin : Représente le retenu d'entrée provenant d'une addition précédente.

Les deux ports de sortie sont les suivants:

S : Représente le résultat de l'addition

Cout : Représente le retenu de l'addition des bits A et B.

Generic:

Les deux variables génériques **TimeS** et **TimeCout** représentent une simulation du temps de propagation qu'un système réel connaîtra.

Comportement de l'additionneur:

Cette section du code représente le comportement de notre additionneur.

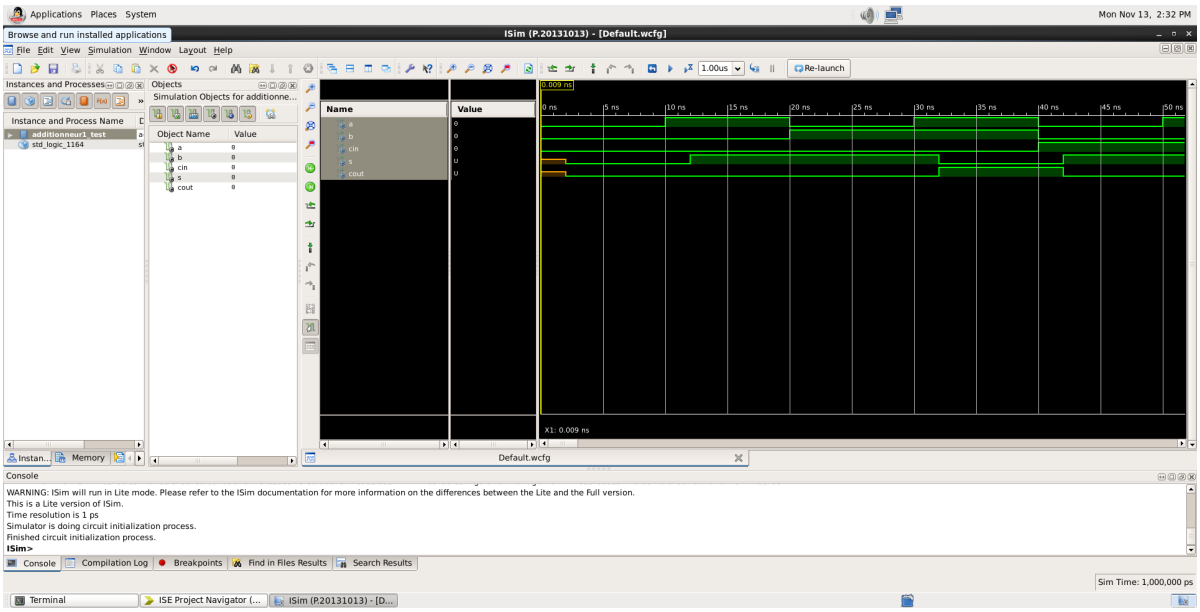
Notre sortie S reçoit le xor des A et B et Cin parceque cette opération représente l'addition sur 1 bits. 1 xor 1 = 0/ 1 xor 1 xor 1 = 1. Le Cout qui recoit 1 si deux ports des 3 d'entrée contiennent la valeur 1 représente correctement le revenu d'une addition de bits.

- Passons par la suite vers l'exposition du code du test de l'additionneur:

Le code du testeur peut être résumé par les étapes suivantes:

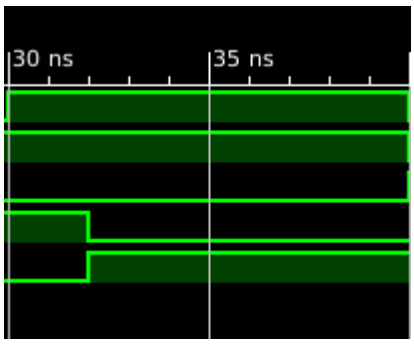
-- On déclare l'entité du test.
--Dans le comportement de l'entité, on commence par déclarer le composant de l'additionneur1, puis on déclare les différents signaux que notre composant va utiliser.
Par la suite, on code chaque entrée et sortie de notre composant au signal créé pour celle-ci. Par conséquent, on aura instancié notre unité sous test.
--Enfin, on simule des différents cas d'addition en donnant des valeurs différents à nos entrées après des périodes de temps différentes.

- Après avoir compris comment notre code fonctionne, on commence une simulation de notre code:



- Expliquons ce qu'on peut voir dans la simulation:

La figure vérifie la validité de notre code. On remarque que notre sortie **S** et notre retenu **Cout** retournent les bonnes valeurs d'une addition de Bits.



Par exemple, dans l'intervalle du temps **[30,40]**ns, on a **A=1**, **B=1**, **Cin=0**, **S=0** et **Cout=1**, ce qui est le résultat normal d'une addition en bits de **1+1+0**.

- Interprétons le temps de propagation:

Les **2ns** de décalage qu'on a ajouté à nos 2 sorties **S** et **Cout** simulent un temps de propagation réel qu'un système réel connaîtra.
On remarque que ce temps de propagation ajoute une erreur temporelle dans chaque résultat de notre addition. Ces erreurs démontrent les limitations et problèmes qu'un développeur devra dépasser lors de l'implémentation d'un système réel.

Revenons à l'exemple de l'intervalle du temps [30,40]ns. Supposons que notre système prend le résultat de notre addition est cultivé 1ns après l'entrée de **A** , **B** et **Cin**. Dans ce cas, la somme de **A=1** et **B=1** et **Cin=0** résultera dans **S=1** et **Cout = 0**, ce qui n'est pas le bon résultat.

Par conséquent, chaque développeur d'un système réel doit prendre en considération ces limitations techniques qui sont présentes dans les systèmes embarqués.

2- Additionneur 4 bits

Après avoir réalisés notre additionneur 1 Bits, passons à la réalisation d'un additionneur 4 bits qui combine 4 des additionneurs 1 bits présentés auparavant.

- Exposant tout d'abord le code entier de notre additionneur:

Additionneur4bits

```
----------------------------------------------------------------------------------
-
-- Company:
-- Engineer:
--
-- Create Date:    16:45:32 10/20/2023
-- Design Name:
-- Module Name:    additionneur_4bits - Behavioral
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
-
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity additionneur_4bits is
    Port ( A4 : in  STD_LOGIC_VECTOR (3 downto 0);
           B4 : in  STD_LOGIC_VECTOR (3 downto 0);
           CIN_4 : in  STD_LOGIC;
           S4 : out  STD_LOGIC_VECTOR (3 downto 0);
           COUT_4 : out  STD_LOGIC);
end additionneur_4bits;

architecture Behavioral of additionneur_4bits is

component add1module is
    Port ( A : in  STD_LOGIC;
```

```
            B : in  STD_LOGIC;
            CY_IN : in  STD_LOGIC;
            S : out  STD_LOGIC;
            CY_OUT : out  STD_LOGIC);
end component;
signal Cout0, Cout1, Cout2: STD_LOGIC;
begin

    additionneur1 : add1module
    port map( A4(0), B4(0), CIN_4, S4(0), Cout0);

    additionneur2 : add1module
    port map( A4(1), B4(1), Cout0, S4(1), Cout1);

    additionneur3 : add1module
    port map( A4(2), B4(2), Cout1, S4(2), Cout2);

    additionneur4 : add1module
    port map( A4(3), B4(3), Cout2, S4(3), COUT_4);


end Behavioral;
```

Additionneur4Bits_test

```
--------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:   16:48:42 10/20/2023
-- Design Name:
-- Module Name:   C:/Users/master/Desktop/TP1/add1/additionneur_4bits_test.vhd
-- Project Name:  add1
-- Target Device:
-- Tool versions:
-- Description:
--
-- VHDL Test Bench Created by ISE for module: additionneur_4bits
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
-- Notes:
-- This testbench has been automatically generated using types std_logic and
-- std_logic_vector for the ports of the unit under test.  Xilinx recommends
-- that these types always be used for the top-level I/O of a design in order
-- to guarantee that the testbench will bind correctly to the post-implementation
-- simulation model.
--------------------------------------------------------------------------------
LIBRARY ieee;
```

```
USE ieee.std_logic_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--USE ieee.numeric_std.ALL;

ENTITY additionneur_4bits_test IS
END additionneur_4bits_test;

ARCHITECTURE behavior OF additionneur_4bits_test IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT additionneur_4bits
    PORT(
         A4 : IN  std_logic_vector(3 downto 0);
         B4 : IN  std_logic_vector(3 downto 0);
         CIN_4 : IN  std_logic;
         S4 : OUT  std_logic_vector(3 downto 0);
         COUT_4 : OUT  std_logic
        );
    END COMPONENT;


   --Inputs
   signal A4 : std_logic_vector(3 downto 0) := (others => '0');
   signal B4 : std_logic_vector(3 downto 0) := (others => '0');
   signal CIN_4 : std_logic := '0';

    --Outputs
   signal S4 : std_logic_vector(3 downto 0);
   signal COUT_4 : std_logic;
   -- No clocks detected in port list. Replace <clock> below with
   -- appropriate port name



BEGIN

    -- Instantiate the Unit Under Test (UUT)
   uut: additionneur_4bits PORT MAP (
          A4 => A4,
          B4 => B4,
          CIN_4 => CIN_4,
          S4 => S4,
          COUT_4 => COUT_4
        );
A4 <= "0000" after 0 ns, "0010" after 200 ns, "0100" after 400 ns, "0110" after
600 ns, "1000" after 800 ns, "1010" after 1000 ns,"1100" after 1200 ns,"1110"
after 1400 ns;
B4 <= "0000" after 0 ns, "0011" after 300 ns, "0101" after 500 ns, "0111" after
700 ns, "1001" after 900 ns, "1011" after 1100 ns,"1101" after 1300 ns,"1111"
after 1500 ns;
CIN_4 <= '0' after 0 ns, '1' after 750 ns;

END;
```

- Expliquons le code.

Grâces aux similarités des deux additionneurs, on se limitera aux différences qui résident entre les deux.

Entité add4bits

Diffère de l'entité **add1Bits** par les ports **A4**, **b4**, et **S4**, qui sont dans ce cas des vecteurs de 4 bits. Chaque Bit des 4 sera attribué à l'un des ports des 4 additionneurs 1 bits qui composent notre système.

Définition du composant add1bits

Etant donné que notre système est composé de 4 additionneurs 1 bit, nous devrons définir ce composant avec ses différents ports, chose qu'on a fait dans l'exercice auparavant.

Signaux intermédiaires

Dans l'architecture comportementale de cette entité, on instancie 3 signaux intermédiaires, qui transporteront les valeurs intermédiaires d'un **Cout** d'un additionneur 1 bit au **Cin** du prochain.

Instanciation et mappages des add1bits

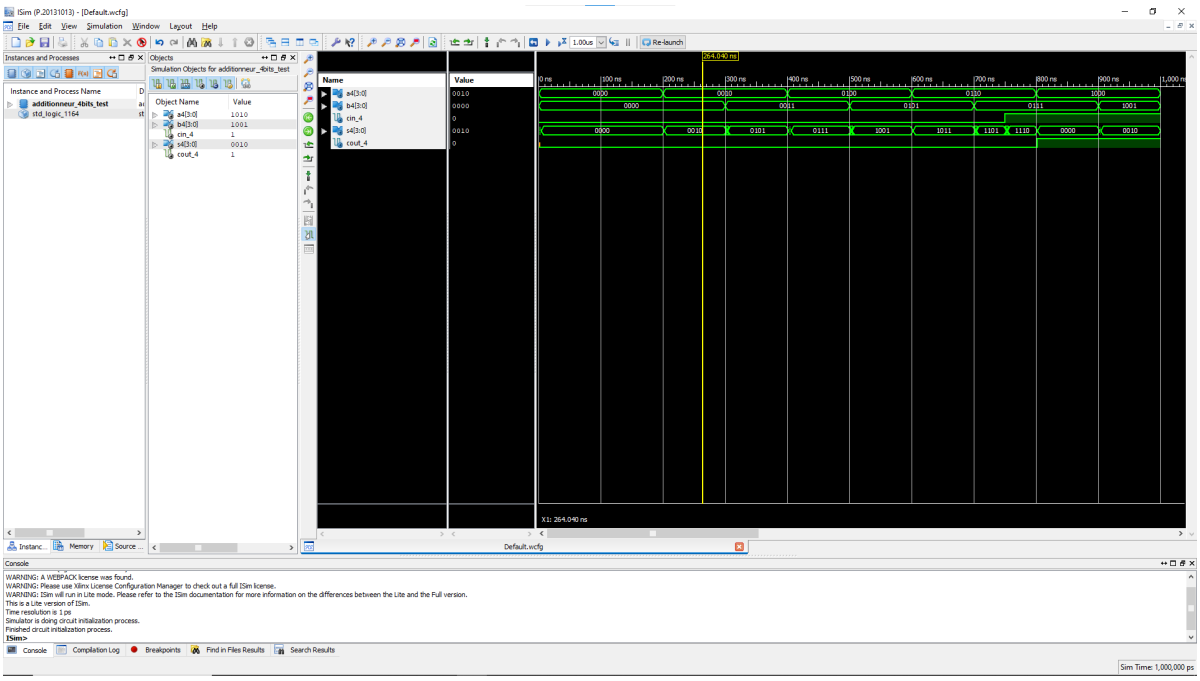
Ayant défini l'architecture de nos additionneurs1bits, nous passons à les instancier. Nous définissons ainsi 4 additionneurs 1 bit, en mappant leurs entrées **Ai** et **Bi** et leurs sorties **Si** aux 4 bits des ports **A4**, **B4** et **S4** respectivement.

Le **Cin** du premier **Cout** du dernier additionneur sont mappés aux **Cin_4** et **Cout4**, tandis que le reste des **Cin(i)** et **Cout(i)** sont mappés aux signaux intermédiaires que nous avons défini (Le **Cout** de l'additionneur i est le **Cin** de l'additionneur i+1).

Changements dans le fichier Test

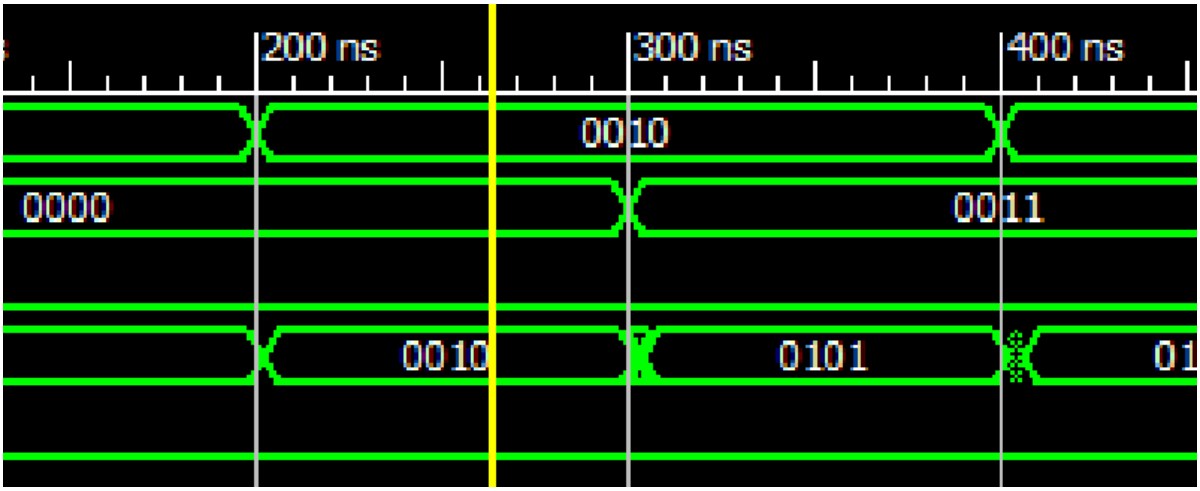
Le fichier Test reste quasiment le même. Nous ne rapportons que les changements nécessaires dans la définition du composant et des signaux (des ports 1bit aux ports 4 bits).

- Simulons notre code:



- Interprétation:

Similairement au cas de l'additionneur 1 bit, notre simulation vérifie la validation de notre code. Prenons de nouveau un exemple pour vérifier cette validité.



L'intervalle **[200,400]ns** nous présente deux exemples d'additions:

-- **[200,300]**: L'addition de 0010 (2) et 0000(0) résulte dans **S** = 0010(2) et **Cout** = 0, ce qui est vrai.

--**[300,400]**: L'addition de 0010(2) et 0011(3) résulte dans **S**=0101(5) et **Cout**=0, ce qui est vrai.

Problème de temps de propagation:

Le problème de décalage simulé demeure dans ce cas. En effet, il s'aggrave, puisque nous avons chaque additionneur1bit Souffre d'un délai de 2ns au niveau de sa sortie.

Par conséquent, notre sortie n'aura sa valeur valide qu'après 4*2 = 8ns. Notre graphe de résultat aura alors une mauvaise représentation, ayant le premier bit du résultat qui recoit une bonne valeur après 2ns, le 2ème après 4ns... On peut voir ce décalage dans notre exemple auparavant, où le résultat **S4** semble changer de valeur hors la ligne de changement de **A4** et **B4**, même avec l'énorme échelle de **100ns**.

Le risque d'erreur monte alors avec le nombre de composants de notre système. Cette affirmation démontre la complicité que nous pourrions rencontrer dans l'établissement d'un système embarqué compliqué qui comportera des milliers de composants.

Il s'avére ainsi raisonnable de traiter le problème depuis les composants élémentaires avant que l'impact de ces petites influences ne s'aggrave avec l'expansion du système.

- Problème de simulation

La simulation proposée au-dessus est efficace dans le cas d'un système simple comme celui qu'on présente. Par contre dans un cas plus compliqué, affecter manuellement des valeurs à chaque port de nos composans afin de vérifier chaque cas possible risque d'être une tâche très compliquée, voir même impossible.

Ainsi, nous optons à utiliser une fonction qui génére et affecte automatiquement des valeurs de test à nos ports. Par conséquent, nous couvrerons tous les cas de test possibles sans avoir à affecter des valeurs manuellement.

--Présentons tout d'abord notre nouveau code amélioré. Etant donné que nous ne changeons que les valeurs affectés pendant la partie test, il est normal que le fichier Add4bits reste intacte. Ainsi, nous ne presenterons que le nouveau fichier Add4Bits_Test:

```
--------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:   12:59:46 11/13/2023
-- Design Name:
-- Module Name:   /home/ise/PartageVM/Partage/TP1_WM/additionneur4_test.vhd
-- Project Name:  TP1
-- Target Device:
-- Tool versions:
-- Description:
--
-- VHDL Test Bench Created by ISE for module: additionneur4
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
-- Notes:
-- This testbench has been automatically generated using types std_logic and
-- std_logic_vector for the ports of the unit under test.  Xilinx recommends
-- that these types always be used for the top-level I/O of a design in order
-- to guarantee that the testbench will bind correctly to the post-implementation
-- simulation model.
--------------------------------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
```

```
--USE ieee.numeric_std.ALL;

ENTITY additionneur4_test IS
END additionneur4_test;

ARCHITECTURE behavior OF additionneur4_test IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT additionneur4
    PORT(
         A4 : IN  std_logic_vector(3 downto 0);
         B4 : IN  std_logic_vector(3 downto 0);
         CIN_4 : IN  std_logic;
         S4 : OUT  std_logic_vector(3 downto 0);
         COUT_4 : OUT  std_logic
        );
    END COMPONENT;

     component gene is
            generic(
                val_init,val_fin,increment:integer;
                delta_t:time
            );

            port (
                sortie:out std_logic_vector(3 downto 0)
            );
     end component;

    component CLOCK is
         generic(DEMI_PERIODE : time);
         port (
            SORTIE : out std_logic
         );
    end component;

    --outputs of genes
    signal sortie1, sortie2: std_logic_vector(3 downto 0);

    --output clock
    signal sortie_clock : std_logic;


   --Inputs
   signal A4 : std_logic_vector(3 downto 0) := (others => '0');
   signal B4 : std_logic_vector(3 downto 0) := (others => '0');
   signal CIN_4 : std_logic := '0';

    --Outputs
   signal S4 : std_logic_vector(3 downto 0);
   signal COUT_4 : std_logic;
   -- No clocks detected in port list. Replace <clock> below with
   -- appropriate port name
```

```
BEGIN

    gen1: gene
    Generic MAP (
        val_init   => 0,
        val_fin    => 15,
        increment  => 1,
        delta_t    => 10 ns)
    PORT MAP( sortie1);

    gen2: gene
    generic MAP (
        val_init   => 0,
        val_fin    => 15,
        increment  => 1,
        delta_t    => 160 ns)
    port map (sortie2);

     clck : CLOCK
     generic map (DEMI_PERIODE => 1280 ns)
     PORT MAP (sortie_clock);

    -- Instantiate the Unit Under Test (UUT)
   uut: additionneur4 PORT MAP (
          A4 => sortie1,
          B4 => sortie2,
          CIN_4 => sortie_clock,
          S4 => S4,
          COUT_4 => COUT_4
        );

END;
```

- Expliquons les nouveautés.

Notre nouveau code test définit un nouveau composant **gene**. Ce composant est un générateur qui prend en entrées une valeur min, une valeur max, un incrément qui lui permet de passer d'une valeur à la prochaine et un intervalle de temps de passage d'une valeur à la prochaine. Il ressort une valeur à 4 bits, qui sera la valeur de nos ports **A4** et **B4**.

En addition, notre nouveau défini un nouveau composant clock, qui prend en entrée une demi période et nous sort en sortie un signal carré. Ce signal représentera une simulation de notre reteni **Cin**.

Nous instancions tout d'abord le générateur des valeurs de **A4**. Puisqu'on travaille sur 4 bits, on lui donne comme valeur minimale 0 et 15 comme valeur maximale. Pour parcourir toutes les valeurs et combinaisons possibles, on lui associe un incrément de 1, et une période choisie de 10ns.

Nous instancions par la suite le générateur des valeurs de B4. Pour les mêmes raisons, on lui donne les valeurs 0, 15 et 1 pour ses valeurs min, max et incrément respectivement. Puisqu'on doit parcourir toutes les combinaisons possibles, pour chaque valeur de B4, le générateur A4 doit avoir parcouru toutes ses valeurs. Ainsi, nous lui donnerons une période de 16***période de gene1** = 16 * 10ns = 160ns.

Enfin, nous instancions le générateur des valeurs de **Cin**. Afin de simuler toutes les combinaisons possibles, on devrait calculer l'addition de toutes les valeurs de **A4** et **B4** sans retenu, et puis avec ce dernier. Ainsi notre période doit être égale à **16*160 = 2560ns** (afin de simuler tous les cas de A4 et B4). Par conséquent, nous donnerons à notre générateur la demi période de **2560/2 = 1280ns**.

--Puisque nous n'avons apporté aucun changement sur ces entités et le fichier **CONV_PKG** d'où elles importent des fonctions utilisées, nous n'expliquerons pas dans cette partie le fonctionnement de cettes dernières. Nous le ferons plus tard dans la partie de l'additionneur 16 bits tout en exposant le code qui est très similaire.

3- Additionneur16Bits

Après la réalisation de l'additionneur4bits, nous appliquerons la même logique (4 Additionneurs 4 bits reliés) pour créer un additionneur 16 bits. Exposons tout d'abord le code:

Additionneur16Bits:

```
----------------------------------------------------------------------------------
-
-- Company:
-- Engineer:
--
-- Create Date:    13:04:11 11/13/2023
-- Design Name:
-- Module Name:    Additionneur16 - Behavioral
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
-
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;
```

```
entity Additionneur16 is
    Port ( A16 : in  STD_LOGIC_VECTOR (15 downto 0);
           B16 : in  STD_LOGIC_VECTOR (15 downto 0);
           CIN_16 : in  STD_LOGIC;
           S16 : out  STD_LOGIC_VECTOR (15 downto 0);
           COUT_16 : out  STD_LOGIC);
end Additionneur16;

architecture Behavioral of Additionneur16 is
    COMPONENT additionneur4
         PORT(
                A4 : IN  std_logic_vector(3 downto 0);
                B4 : IN  std_logic_vector(3 downto 0);
                CIN_4 : IN  std_logic;
                S4 : OUT  std_logic_vector(3 downto 0);
                COUT_4 : OUT  std_logic
              );
         END COMPONENT;
    signal cout0, cout1, cout2 : STD_LOGIC;
begin
        add1 : additionneur4
        port map( A16(3 downto 0), B16(3 downto 0), CIN_16, S16(3 downto 0),
cout0);

        add2 : additionneur4
        port map( A16(7 downto 4), B16(7 downto 4), cout0, S16(7 downto 4),
cout1);

        add3 : additionneur4
        port map( A16(11 downto 8), B16(11 downto 8), cout1, S16(11 downto 8),
cout2);

        add4 : additionneur4
        port map( A16(15 downto 12), B16(15 downto 12), cout2, S16(15 downto 12),
COUT_16);

end Behavioral;
```

En suivant la même logique qu'auparavant, on change les ports de notre additionneur à des vecteurs de 16 bits. Nous définissons par la suite le composant de l'additionneur 4 bits, et on map les ports de l'additionneur 16 bits à celles des additionneurs 4 bitsn tout en définissant des signaux intermédiaires pour les **Cin** et **Cout**.

Avant de passer à Additionneur16bits_test, exposant tout d'abord l'entité génératrice des valeurs utilisées par notre simulateur.

```
----------------------------------------------------------------------------------
-
-- Company:
-- Engineer:
--
-- Create Date:    15:00:48 11/13/2023
-- Design Name:
-- Module Name:    gene_16 - Behavioral
-- Project Name:
```

```
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
-
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

use work.conv_pkg.all;


-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity gene_16 is
    generic(val_init,val_fin,increment:integer;delta_t:time);
    Port ( sortie : out  STD_LOGIC_VECTOR (15 downto 0));
end gene_16;

architecture Behavioral of gene_16 is

      signal sortie_entiere:integer:=val_init;

begin


    general:process

    begin

     wait for delta_t;         --maintien val_init
     if sortie_entiere+increment>val_fin  --increment possible?
        then
        sortie_entiere<=val_init;     --retour val_init
        else
        sortie_entiere<=sortie_entiere +increment; --increment
     end if;

    end process general;

sortie<=ENTIER_TO_BIT_16(sortie_entiere);  --affectation concurente

end Behavioral;
```

Cette entité prend en entrées 3 entiers (Valeur_minimale, Valeur_maximale, incrément) et une variable temps delta_t. Elle ressort les valeurs de simulation de nos **A16**, **B16** et **Cin_16**.

Son fonctionnement est comme suit:

-- Le générateur attend le déroulement de **delta_t**.

-- Il vérifie par la suite si une incrémentation est possible (valeur courant inférieure strictement à la valeur finale), si oui il incrémente. Si non, le générateur réinitialise la valeur de sortie (ce qui sera utile lorsqu'on veut simuler toutes les combinaisons (A16 devra parcourir toutes ses valeurs 3 fois)).

Nous remarquons que cette entité utilise une fonction Entier_to_bit_16, exposant la:

```
function ENTIER_TO_BIT_16(N: in INTEGER) return std_logic_vector is
        variable V:std_logic_vector(15 downto 0);
        variable X:INTEGER:=0;
        begin
        X:=N;
        for I in V'REVERSE_RANGE loop
        if X rem 2=0 then
            V(I):='0';
        else
            V(I):='1';
        end if;
        X:=X/2;
        end loop;
        return V;
end ENTIER_TO_BIT_16;
```

Comme son nom l'indique, prend un entier et le transforme en binaire. Le changement qu'on a apporté à la fonction Entier_to_Bit donnée est de transformer la variable de sortie en 16 bits à la place de 4.

La fonction rentre dans une boucle à partir du bits le plus fort de notre vecteur. Elle applique par la suite le principe des divisions consécutives qu'on utilise pour transformer un entier en son équivalent binaire, en prenant le reste de la division par 2 à chaque itération et l'affectant au bit courant. Vers la dernière itération, on aura l'équivanlent binaire en 16 bits de notre entier. Une version 4 bits a été utilisé dans l'additionneur 4 bits.

Passons par la suite à l'entité clock;

```
-- generateur d'horloge symetrique pour test
-- parametre generique: demi periode

library IEEE;
use IEEE.std_logic_1164.all;

entity CLOCK is
 generic(DEMI_PERIODE : time);
 port (SORTIE : out std_logic);
end CLOCK;

architecture A_CLOCK of CLOCK is
```

```
begin
 general:process
  variable PERIODE :time;

  begin
   PERIODE := DEMI_PERIODE * 2;
   SORTIE<='0','1' after DEMI_PERIODE;
   wait for PERIODE;
  end process general;
end A_CLOCK;
```

Le fonctionnement de cette entité est simple. Elle sort une valeur de 0 pendant un delta_t = demi_période donnée en entrée, et 1 pendant la 2ème demi période. cette même entité a été utlisée pour générer le retenu de l'additionneur 4 bits.

Après l'expliquation de nos générateurs, passons à notre test:

```
--------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:   14:40:52 11/13/2023
-- Design Name:
-- Module Name:   /home/ise/PartageVM/Partage/TP1_WM/additionneur16_test.vhd
-- Project Name:  TP1
-- Target Device:
-- Tool versions:
-- Description:
--
-- VHDL Test Bench Created by ISE for module: Additionneur16
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
-- Notes:
-- This testbench has been automatically generated using types std_logic and
-- std_logic_vector for the ports of the unit under test.  Xilinx recommends
-- that these types always be used for the top-level I/O of a design in order
-- to guarantee that the testbench will bind correctly to the post-implementation
-- simulation model.
--------------------------------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--USE ieee.numeric_std.ALL;

ENTITY additionneur16_test IS
END additionneur16_test;
```

```
ARCHITECTURE behavior OF additionneur16_test IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT Additionneur16
    PORT(
         A16 : IN  std_logic_vector(15 downto 0);
         B16 : IN  std_logic_vector(15 downto 0);
         CIN_16 : IN  std_logic;
         S16 : OUT  std_logic_vector(15 downto 0);
         COUT_16 : OUT  std_logic
        );
    END COMPONENT;

     component gene_16 is
            generic(
                val_init,val_fin,increment:integer;
                delta_t:time);

            port (
                sortie:out std_logic_vector(15 downto 0));
     end component;

    component CLOCK is
         generic(DEMI_PERIODE : time);
         port (
            SORTIE : out std_logic
         );
    end component;


   --Inputs
   signal A16 : std_logic_vector(15 downto 0) := (others => '0');
   signal B16 : std_logic_vector(15 downto 0) := (others => '0');
   signal CIN_16 : std_logic := '0';

    --Outputs
   signal S16 : std_logic_vector(15 downto 0);
   signal COUT_16 : std_logic;


    --outputs of genes
    signal sortie1, sortie2: std_logic_vector(15 downto 0);

    --output clock
    signal sortie_clock : std_logic;




BEGIN

    gen1: gene_16
    Generic MAP (
        val_init   => 0,
```

```
        val_fin    => 65535,
        increment  => 1,
        delta_t    => 1 ns)
    PORT MAP( sortie1);

    gen2: gene_16
    generic MAP (
        val_init   => 0,
        val_fin    => 65535,
        increment  => 1,
        delta_t    => 65535 ns)
    port map (sortie2);

     clck : CLOCK
     generic map (DEMI_PERIODE => 2147418112.5 ns)
     PORT MAP (sortie_clock);

    -- Instantiate the Unit Under Test (UUT)
   uut: Additionneur16 PORT MAP (
          A16 => sortie1,
          B16 => sortie2,
          CIN_16 => sortie_clock,
          S16 => S16,
          COUT_16 => COUT_16
        );


END;
```

Ce test respecte exactement la même logique utilisée dans l'additionneur 4 bits, nous ne changeons que les valeurs d'entrée de nos générateurs pour couvrir tous les cas.

La simulation du code révèle la validité de ce dernier, mais relève au même temps la problématique soulignée auparavant. Dans ce cas, le retard s'accumulera à 32 ns pour le dernier bit de sortie, ce qui est énorme et peut engendrer des erreurs de calculs plus fréquemment.

Conclusion

Ce premier TP a constitué une immersion captivante dans l'univers complexe de la programmation des systèmes embarqués. Il a offert une opportunité concrète d'évaluer nos compétences en VHDL à travers des projets concrets. De plus, cette expérience a éclairé les difficultés inhérentes à la programmation de systèmes complexes et réels, nous invitant à anticiper et surmonter les limitations qui peuvent se présenter. Cette exploration initiale a jeté les bases pour une compréhension approfondie des défis à relever dans le domaine de la programmation embarquée.

TP2 : Compteur

L'objectif de ce deuxième TP est de concevoir un compteur binaire en VHDL qui répond à un cahier de charges donné. Le compteur devait prendre en charge différentes fonctionnalités telles que le maintien de la valeur, le chargement d'une valeur, la gestion de limites, et la détection d'overflow.

Commençons tout d'abord par la création de notre entité compteur. Le code proposé est le suivant:

```
----------------------------------------------------------------------------------
-
-- Company:
-- Engineer:
--
-- Create Date:    15:51:49 11/13/2023
-- Design Name:
-- Module Name:    compteur - Behavioral
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
-
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity compteur is
    Port ( Limite1    : in  STD_LOGIC_VECTOR (7 downto 0);
           Limite2    : in  STD_LOGIC_VECTOR (7 downto 0);
           Chargement : in  STD_LOGIC_VECTOR (7 downto 0);
           hold       : in  STD_LOGIC;  --if hold=1, sortie garde sa valeur
           load_count : in  STD_LOGIC;  -- if =1, sortie=chargement, sinon
incremente
           load_limit : in  STD_LOGIC;  --if =1, sel_limit, else limit constant
           sel_limit  : in  STD_LOGIC;  -- if=1, limit=limit1, else limit=limit2
           clock      : in  STD_LOGIC;
           sortie     : out STD_LOGIC_VECTOR (7 downto 0);
```

```
           overflow   : out STD_LOGIC);  --passe à 1 si la limite est atteinte
chargement=limiti
end compteur;

architecture Behavioral of compteur is
    signal counter_value : STD_LOGIC_VECTOR(7 downto 0);
    signal limit_value   : STD_LOGIC_VECTOR(7 downto 0);
    signal overflow_flag : STD_LOGIC := '0';

begin
    process (clock)
    begin
        if rising_edge(clock) then
            -- Vérifier le signal hold
            if hold = '0' then
                counter_value <= Chargement;
                        end if;
                -- Vérifier le signal load_count
            if load_count = '1' then
                counter_value <= Chargement;
            else
                -- Incrémenter le compteur
                counter_value <= counter_value + 1;

                -- Vérifier les signaux load_limit et sel_limit
                if load_limit = '1' then
                    limit_value <= Chargement;
                elsif sel_limit = '1' then
                    limit_value <= Limite1;
                else
                    limit_value <= Limite2;
                end if;

                -- Vérifier le dépassement
                if counter_value = limit_value then
                    overflow_flag <= '1';
                else
                    overflow_flag <= '0';
                end if;
            end if;
        end if;
    end process;

    -- Assigner la valeur du compteur à la sortie
    sortie <= counter_value;
    overflow <= overflow_flag;

end Behavioral;
```

Nous avons commencé tout d'abord par la conception d'un compteur binaire simple,. Ensuite, nous avons ajouté les différentes fonctionnalités demandées.

Ainsi, nous avons créé une entité VHDL appelée compteur avec les ports spécifiés dans le cahier des charges. La conception interne du compteur utilise un processus déclenché par le front montant de l'horloge pour mettre à jour la valeur du compteur en fonction des signaux d'entrée.

Afin de tester notre code, nous avons créé un testBench qui valide le fonctionnement fiable à notre cahier de charges. Ce test bench associe des différentes valeurs à nos entrées, et vérifie la sortie engendrée. Le code du test bench est le suivant:

```
----------------------------------------------------------------------------------
-
-- Company:
-- Engineer:
--
-- Create Date:    16:51:49 11/13/2023
-- Design Name:
-- Module Name:    compteur - Behavioral
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
-
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity compteur_test is
end compteur_test;

architecture testbench of compteur_test is
    -- Déclarez les signaux du testbench
    signal Limite1_tb, Limite2_tb, Chargement_tb : STD_LOGIC_VECTOR(7 downto 0);
    signal hold_tb, load_count_tb, load_limit_tb, sel_limit_tb, clock_tb :
STD_LOGIC;
    signal sortie_tb : STD_LOGIC_VECTOR(7 downto 0);
    signal overflow_tb : STD_LOGIC;

    -- Instancier le composant compteur
    component compteur
        Port ( Limite1    : in  STD_LOGIC_VECTOR (7 downto 0);
               Limite2    : in  STD_LOGIC_VECTOR (7 downto 0);
               Chargement : in  STD_LOGIC_VECTOR (7 downto 0);
               hold       : in  STD_LOGIC;
               load_count : in  STD_LOGIC;
```

```
            load_limit : in  STD_LOGIC;
            sel_limit  : in  STD_LOGIC;
            clock      : in  STD_LOGIC;
            sortie     : out STD_LOGIC_VECTOR (7 downto 0);
            overflow   : out STD_LOGIC);
    end component;

    -- Générez le signal d'horloge
    constant clock_period : time := 10 ns; -- Définissez la période de l'horloge
    signal clock_gen : STD_LOGIC := '0';

begin

    uut : compteur
        port map (
            Limite1 => Limite1_tb,
            Limite2 => Limite2_tb,
            Chargement => Chargement_tb,
            hold => hold_tb,
            load_count => load_count_tb,
            load_limit => load_limit_tb,
            sel_limit => sel_limit_tb,
            clock => clock_gen,
            sortie => sortie_tb,
            overflow => overflow_tb
        );

    -- Processus pour générer l'horloge
    clock_process: process
    begin
        while now < 500 ns loop
            clock_gen <= not clock_gen;
            wait for clock_period / 2;
        end loop;
        wait;
    end process;

    -- Processus pour appliquer des scénarios de test
    stimulus_process: process
    begin

        hold_tb <= '0';
        load_count_tb <= '0';
        load_limit_tb <= '0';
        sel_limit_tb <= '0';
        clock_tb <= '0';
        wait for 20 ns;

        hold_tb <= '1';
        wait for 20 ns;

        hold_tb <= '0';
        load_count_tb <= '1';
        wait for 20 ns;

        wait;
```

```
    end process;

end testbench;
```

On simule dans ce test la réaction de notre face aux changement des différentes valeurs des ports de notre entité. Le signal de sortie vérifie la validité et fiabilité de notre code avec le cahier de charges proposé.

Conclusion

Ce deuxième TP a significativement enrichi notre expérience en programmation des systèmes embarqués à travers l'utilisation du langage VHDL. Cette étape nous a familiarisés avec la mise en œuvre de processus réactifs déclenchés par des événements spécifiques. L'absence de directives détaillées dans le Test Bench a constitué une opportunité d'explorer la génération autonome de scénarios de test. Cette démarche a mis en lumière l'importance de la création de tests robustes pour évaluer la validité du code et sa conformité aux exigences spécifiées dans le cahier des charges.