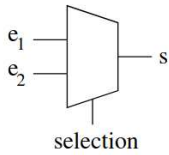


Architecture des ordinateurs - TD 1

On désire implanter et tester un multiplexeur 2 bits vers 1 en VHDL et ceci en utilisant les différentes vues abordées en cours (vue structurelle, vue flot de données, vue comportementale).

L'interface du multiplexeur est la suivante



s vaut e1 si selection = '0'
s vaut e2 si selection = '1'

Nous disposons de 3 composants porte et, porte ou et inverseur dont voici les interfaces :

```

-- temps de traversée : 2 ns
entity porte_et is
  port ( i1, i2 : in std_logic;
         s : out std_logic );
end porte_et;

-- temps de traversée : 2 ns
entity porte_ou is
  port ( i1, i2 : in std_logic;
         s : out std_logic );
end porte_ou;

-- temps de traversée : 1 ns
entity inverseur is
  port ( i : in std_logic;
         s : out std_logic );
end inv;
```

1. donner l'interface vhdl du multiplexeur

```

1 entity multiplexeur is
2   port ( e1, e2, selection : in std_logic;
3         s : out std_logic;
4   );
5 end multiplexeur
```

2. décrire les architectures de ce composant selon les trois vues

```

1 architecture dataflow1 of multiplexeur is
2   begin
3     s <= e1 after 5 ns when ( selection = '0') else
4       e2 after 5 ns when ( selection = '1' ) else
5       'X' after 5 ns;
6   end dataflow1
```

Ce qui est factorisable avec un switch case vhdl

```

1 architecture dataflow1 of multiplexeur is
2   begin
3     with selection select
4       s <= e1 after 5 ns when '0',
5         e2 after 5 ns when '1',
6         'X' after 5 ns when others;
7   end dataflow1
8
```

Le fait d'avoir défini la structure architectural de notre composant permet de factoriser du code, dans les faits si on voulait on aurait pu écrire tout le comportement du composant avec ces signaux interne (faut être un enclulé pour)

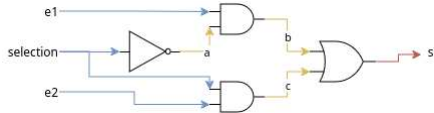
```

1 architecture dataflow1 of multiplexeur is
2   signal a, b, c : std_logic
3   begin
4     with selection select
5       s <= b or c after 2 ns
6       a <= not e1 after 1 ns
7       ...
8
9   end dataflow1
10
```

On a comme description comportemental du composant :

```

1 architecture behavioural of mymultiplexeur is
2   begin
3     process(e1, e2, selection)
4       begin
5         s <= (e1 and not selection) or (e2 and selection) after 5ns;
6       end process;
7   end;
8
9
```



```

1 architecture structural of multiplexeur is
2
3   -- Définition des composants
4   component porte_et
5     port(i1, i2 : in std_logic;
6         s : out std_logic);
7   end component;
8   component inverseur
9     port(i1 : in std_logic;
10        s : out std_logic);
11  end component
12  component porte_ou
13    port(i1, i2 : in std_logic;
14        s : out std_logic);
15  end component;
16
17  --- Définition signal
18  signal a, b, c: std_logic;
19
20  begin
21
22    -- Architecture du composant et signaux intermédiaire.
23    u1 : inverseur port map (selection, a); -- équivalent à port map (i1 => se
24    u2 : porte_et port map (i1 => e1, i2 => a, s => b);
25    u3 : porte_et port map (i1 => selection, i2 => e2, s => c);
26    u4 : porte_ou port map (i1 => b, i2 => c, s);
27
28  end structural;
```

Equation logique:
s = e1 . !selection + e2.selection

Ainsi on a le dataflow suivant : (Attention a pas oublier qu'on est dans le domaine concurant pour ce genre de description!)

```

1 architecture dataflow1 of multiplexeur is
2   begin
3     s <= (e1 and (not selection)) or (e2 and selection) after 5 ns
4   end dataflow1
```

On peut aussi faire avec des if :

Architecture des ordinateurs - TD 2

Objectifs

Mise en oeuvre de composants cadencés par une horloge.

1. Retour sur les process synchrones du cours et leur intégration dans un composant Compteur

1.1 Process synchrone sur une horloge

```
1 process (clk)
2   variable cpt_aux : std_logic_vector(3 downto 0) := (others => '0');
3 begin
4   if (clk = '1') then
5     cpt_aux := cpt_aux + 1;
6     cpt <= cpt_aux;
7   end if;
8 end process;
```

1.2 Process synchrone sur une horloge avec reset asynchrone

```
1 process (clk, reset)
2   variable cpt_aux : std_logic_vector(3 downto 0) := (others => '0');
3 begin
4   if (reset = '0') then
5     cpt_aux := (others => '0');
6     cpt <= cpt_aux;
7   elsif (rising_edge(clk)) then
8     cpt_aux := cpt_aux + 1;
9     cpt <= cpt_aux;
10  end if;
11 end process;
```

1.3 Composant Compteur avec process synchrone

2. Adaptation du composant Compteur

Modifiez l'interface et le fonctionnement du compteur en rajoutant un signal en sortie qui indique quand le compteur passe à zéro.

Solution avec variable

```
1 entity compteur is
2   port ( clk, reset : in std_logic;
3         cpt : out std_logic_vector(3 downto 0);
4         carry : out std_logic;
5   );
6 end compteur;
7
8 architecture Behavioral of compteur is
9   signal cpt_aux : std_logic_vector(3 downto 0);
10 begin
11   carry <= '1' when (cpt_aux = 0) else '0';
12   process (clk, reset)
13   begin
14     if (reset = '0') then
15       cpt_aux <= (others => '0');
16       cpt <= cpt_aux;
17     elsif (rising_edge(clk)) then
18       cpt_aux <= cpt_aux + 1;
19       cpt <= cpt_aux;
20     end if;
21   end process;
22 end compteur_behavioural;
```

Cependant, `cpt` a un front montant de l'horloge de retard par rapport à `cpt_aux`. Pour régler ce problème, on effectue les modifications suivantes (lignes 17 et 19) :

```
- cpt <= cpt_aux;
+ cpt <= (others => '0')

- cpt <= cpt_aux;
+ cpt <= cpt_aux + 1;
```

Mais on peut encore faire mieux !

```
entity compteur is
  port ( clk, reset : in std_logic;
        cpt : out std_logic_vector(3 downto 0);
        carry : out std_logic;
  );
end compteur;

architecture Behavioral of compteur is
begin
  process (clk, reset)
    variable cpt_aux : std_logic_vector(3 downto 0) := (others => '0');
  begin
    if (reset = '0') then
      carry <= '0';
      cpt_aux := (others => '0');
      cpt <= cpt_aux;
    elsif (rising_edge(clk)) then
      cpt_aux := cpt_aux + 1;
      cpt <= cpt_aux;
      if (cpt_aux = 0) then
        carry <= '1';
      else
        carry <= '0';
      end if;
    end if;
  end process;

end compteur_behavioural;
```

Solution sans variable

On pourrait aussi faire comme suit :

```
1 entity compteur is
2   port ( clk, reset : in std_logic;
3         cpt : out std_logic_vector(3 downto 0);
4         carry : out std_logic;
5   );
6 end compteur;
7
8 architecture Behavioral of compteur is
9   signal cpt_aux : std_logic_vector(3 downto 0);
10 begin
11   carry <= '1' when (cpt_aux = 0) and (reset = '1') else '0';
12   cpt <= cpt_aux;
13   process (clk, reset)
14   begin
15     if (reset = '0') then
16       cpt_aux <= (others => '0');
17     elsif (rising_edge(clk)) then
18       cpt_aux <= cpt_aux + 1;
19     end if;
20   end process;
21 end compteur_behavioural;
```

3. Un Compteur synchrone avec commande

Un compteur donne en sortie une valeur entière à chaque front montant d'une horloge. Les entiers sont représentés par un nombre fixé de bits, nombre donné par un paramètre générique. Une commande en entrée permet de modifier le comportement du compteur ; cette commande a cinq valeurs :

1. *STAY* maintient la valeur
2. *LOAD* permet de charger une valeur (donnée en entrée) dans le compteur
3. *INC* incrémente la valeur du compteur en recommençant à zéro quand la valeur maximale a été atteinte
4. *DEC* décrémente la valeur en recommençant à la valeur maximale quand zéro a été atteint
5. *RAZ* remet à zéro la valeur du compteur

La valeur de la commande est prise en compte sur chaque front montant de l'horloge (compteur synchrone). L'état du compteur correspond en fait à cette valeur de la commande. Quand la commande est *DEC* ou *INC* et que la valeur en sortie atteint zéro, une retenue est mise à '1' en sortie du compteur puis repasse à '0' dès que la valeur redevient non nulle.

Question : Ecrivez le code correspondant à l'interface et à une architecture de ce compteur.

Utilisation d'un package pour définir un type utilisable dans l'entité

Fichier `pack_compteur.vhd` :

```

package pack_compteur is
    type t_cmd is (RAZ, LOAD, INC, DEC, STAY);
end package;

**Fichier `compteur.vhd` : **
```vhd
library IEEE;
use IEEE.std_logic_1164.all;
use work.pack_compteur.all;

entity compteur is
 generic (
 cpt_width : natural := 4;
);

 port (clk: in std_logic;
 cmd: in t_cmd;
 cpt: out std_logic_vector(cpt_width-1 downto 0);
 incpt: in std_logic_vector(cpt_width-1 downto 0);
 carry: out std_logic;
);
end compteur;

architecture Behavioral of compteur is
begin

 process (clk)
 variable cpt_aux : std_logic_vector(cpt_width downto 0) :=
 (others => '0');
 variable changed : boolean;
 begin
 changed := false;

 if (rising_edge(clk)) then
 if cmd == RAZ then
 cpt_aux := (others => '0'); -- RAZ
 else if cmd == LOAD then
 cpt_aux := incpt; -- LOAD
 else if cmd == INC then
 cpt_aux := cpt_aux + 1; -- INC
 changed := true;
 else if cmd == DEC then
 cpt_aux := cpt_aux - 1; -- DEC
 changed := true;
 else if cmd == STAY then
 cpt_aux := cpt_aux; -- STAY
 end if;

 cpt <= cpt_aux;
 end if;

 if changed and cpt_aux == 0 then
 carry <= '1';
 end if;
 end process;
end Behavioral;

```

# Architecture des ordinateurs - TD 3 : Afficher les 8 Sept-Segments

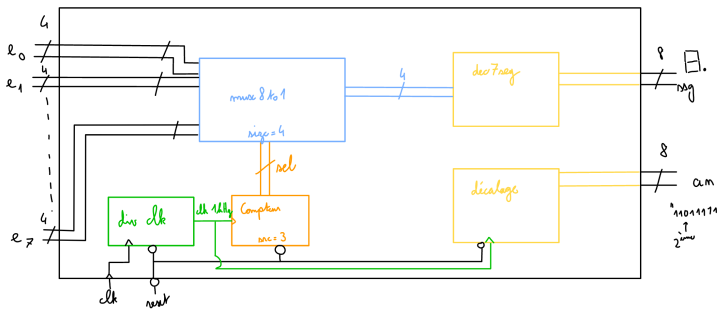
Concevez, développez et testez (en TP) un composant qui permet d'afficher 8 valeurs différentes sur les 8 Sept-Segments.

Le principe est d'afficher de manière cyclique chacune des valeurs en sélectionnant chaque 7-Segments pendant une période suffisamment rapide pour duper l'oeil.

Une horloge à 1 kHz permet d'avoir un affichage correct.

Trois versions sont à développer :

- 1. une vue structurelle (que des sous-composants reliés les uns aux autres)



```
begin

 DivClk : diviseurClk
 generic map (facteur => 100000) -- pas sûr de celle là pour le coup
 port map (clk => clk, reset => reset, nclk => iclk);

 Cpt : compteur
 generic map (size => 3)
 port map (clk, reset, c);

 Mux : mux8_to_1
 generic map(size => 4)
 port map (hex0, hex1, hex2, hex3, hex4, hex5, hex6, hex7, sel, selVal);

 Deco : dec7seg
 port map (selVal, ssg);

 Deca : decalage
 port map(clk, reset, an);

end structural;
```

- 2. une vue comportementale avec un process qui va permettre de compter,

```
entity All_sept_segments is
 port (
 clk, reset: in std_logic;
 e0, e1, e2, e3, e4, e5, e6, e7: in std_logic_vector(3 downto 0);
 ssg, an: out std_logic_vector(7 downto 0);
)
end All_sept_segments;

architecture structural of All_sept_segments is
 -- Rappel des composants

 component diviseurClk
 generic (facteur : natural);
 port (
 clk,reset : in std_logic;
 nclk : out std_logic
);
 end component;

 component dec7seg
 port (
 v : in std_logic_vector(3 downto 0);
 seg : out std_logic_vector(7 downto 0)
);
 end component;

 component compteur
 generic (size : natural := 4);
 port (
 clk : in std_logic;
 reset : in std_logic;
 cpt : out std_logic_vector(size=1 downto 0)
);
 end component;

 component mux8_to_1
 generic (size: natural := 4) ;
 port (e0 , e1 , e2 , e3 ,
 e4 , e5 , e6 , e7 : in std_logic_vector(size=1 downto 0);
 sel : in std_logic_vector(2 downto 0);
 s : out std_logic_vector (size=1 downto 0)
);
 end component;

 component decalage
 port (
 clk : in std_logic;
 reset : in std_logic;
 v : out std_logic_vector(7 downto 0)
);
 end component;

 -- Signaux internes
 signal iclk : std_logic;
 signal selVal : std_logic_vector(3 downto 0);

 architecture behavioral of All_sept_segments is
 -- Signaux
 signal clk1kHz : std_logic;
 signal s : std_logic_vector(3 downto 0);
 signal cpt : natural;

 begin

 dclk : diviseurClk
 generic map (100000)
 port map (clk, reset, clk1kHz);

 process (clk1kHz, reset)
 -- Variables
 -- Constantes
 begin
 if reset = '0' then
 -- RESET : on reset les compteurs, on réinitialise l'écran
 ssg <= (others => '0');
 an <= (others => '1');
 cpt <= 0;
 elsif (rising_edge(clk1kHz)) then
 cpt <= (cpt + 1) mod 8;
 -- ne marche que parce que 8 est une puissance de 2 !
 case cpt is
 when 0 =>
 s <= e0;
 -- an <= "11111110";
 when 1 =>
 s <= e1;
 -- an <= "11111101";
 when 2 =>
 s <= e2;
 -- an <= "111111011";
 when 3 =>
 s <= e3;
 -- an <= "11110111";
 when 4 =>
 s <= e4;
 -- an <= "11101111";
 when 5 =>
 s <= e5;
 -- an <= "11011111";
 when 6 =>
 s <= e6;
 -- an <= "10111111";
 when 7 =>
 s <= e7;
 -- an <= "01111111";
 when others => noop; -- ???
 end case;

 -- ou mieux, on fait après le switch case :
 an <= (others => '1');
 an(cpt) <= '0';
 end if;
 end process;
 end behavioral;
```

```
architecture behavioral of All_sept_segments is
 -- Signaux
 signal clk1kHz : std_logic;
 signal s : std_logic_vector(3 downto 0);
 signal cpt : natural;

 begin

 dclk : diviseurClk
 generic map (100000)
 port map (clk, reset, clk1kHz);

 process (clk1kHz, reset)
 -- Variables
 -- Constantes
 begin
 if reset = '0' then
 -- RESET : on reset les compteurs, on réinitialise l'écran
 ssg <= (others => '0');
 an <= (others => '1');
 cpt <= 0;
 elsif (rising_edge(clk1kHz)) then
 cpt <= (cpt + 1) mod 8;
 -- ne marche que parce que 8 est une puissance de 2 !
 case cpt is
 when 0 =>
 s <= e0;
 -- an <= "11111110";
 when 1 =>
 s <= e1;
 -- an <= "11111101";
 when 2 =>
 s <= e2;
 -- an <= "111111011";
 when 3 =>
 s <= e3;
 -- an <= "11110111";
 when 4 =>
 s <= e4;
 -- an <= "11101111";
 when 5 =>
 s <= e5;
 -- an <= "11011111";
 when 6 =>
 s <= e6;
 -- an <= "10111111";
 when 7 =>
 s <= e7;
 -- an <= "01111111";
 when others => noop; -- ???
 end case;

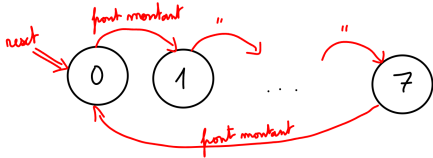
 -- ou mieux, on fait après le switch case :
 an <= (others => '1');
 an(cpt) <= '0';
 end if;
 end process;
 end behavioral;
```

```

end process;
end behavioral;

```

3. une vue comportementale avec un process qui va implémenter un automate à états (fsm - finite state machine).



```

end case;
end if;
end process;
end finite_state_machine;

```

On dispose des composants en annexe. On peut en rajouter d'autres dont on précisera les interfaces.

## 2 Annexes

### 2.1 diviseur d'horloge

```

entity diviseurClk is
-- facteur : ratio entre la fréquence de l'horloge origine et celle
-- de l'horloge générée
-- ex : 100 MHz => 1Hz : 100 000 000
-- ex : 100 MHz => 1kHz : 100 000
generic (facteur : natural);
port (
 clk,reset : in stdlogic;
 nclk : out stdlogic
);
end diviseurClk;

```

### 2.2 Afficheur-7segments

```

entity dec7seg is
port (
 v : in std_logic_vector(3 downto 0);
 seg : out std_logic_vector(7 downto 0)
);
end dec7seg;

```

### 2.3 compteur

```

entity compteur is
generic (size : natural := 4);
port (
 clk : in std logic;
 reset : in std logic;
 cpt : out std logic vector(size=1 downto 0)
);
end compteur;

```

### 2.4 mux8\_to\_1

```

architecture finite_state_machine of All_sept_segments is
-- Types
type t_etat is (zero, un, deux, trois, quatre, cinq, six, sept);

-- Signaux
signal etat : t_etat;
signal clk1kHz : std_logic;
signal s : std_logic_vector(3 downto 0);

begin

dclk : diviseurClk
 generic map (100000)
 port map (clk, reset, clk1kHz);

process(clk1kHz, reset)
begin
 if(reset = '0') then
 etat <= zero;
 s <= (others => '0');
 an <= (others => '1');
 elsif (rising_edge(clk1kHz)) then
 case etat is
 when zero =>
 s <= e0;
 an <= "11111110";
 etat <= un;
 when un =>
 s <= e1;
 an <= "11111101";
 etat <= deux;
 when deux =>
 s <= e2;
 an <= "11111011";
 etat <= trois;
 when trois =>
 s <= e3;
 an <= "11110111";
 etat <= quatre;
 when quatre =>
 s <= e4;
 an <= "11101111";
 etat <= cinq;
 when cinq =>
 s <= e5;
 an <= "11011111";
 etat <= six;
 when six =>
 s <= e6;
 an <= "10111111";
 etat <= sept;
 when sept =>
 s <= e7;
 an <= "01111111";
 etat <= zero;
 when others => noop; -- ???
 end case;
 end if;
end process;
end finite_state_machine;

```

```

entity mux8_to_1 is
generic (size : natural := 4) ;
port (e0 , e1 , e2 , e3 ,
 e4 , e5 , e6 , e7 : in std_logic_vector(size=1 downto 0);
 sel : in std_logic_vector(2 downto 0);
 s : out std_logic_vector (size=1 downto 0)
);
end mux8_to_1;

```

### 2.5 decalage

```

entity decalage is
-- v contient un seul '0' (utilisation : anode == segment allumé)
-- à chaque front montant de l'horloge,
-- la valeur de v est décalée cycliquement d'une position vers la gauche
-- "11101111" => "11011111"
-- "01111111" => "11111110"
port (
 clk : in std_logic;
 reset : in std_logic;
 v : out std_logic_vector(7 downto 0)
);
end decalage;

```

# Architecture des ordinateurs - TD 4 : Automates

Sur un exemple, nous allons voir comment coder le fonctionnement synchrone d'un composant en utilisant un automate à états.

## 1 Composant d'émission d'un octet

Nous souhaitons développer un composant synchrone sur une horloge **clk** dont le rôle est d'émettre un octet selon un protocole série.

Ce composant est par défaut au repos (ligne série **txd** à '1'), en attente d'un ordre d'émission.

Sur le premier front montant de **clk** où il voit le signal **en** (pour *enable*), à '1',

- 1. il stocke l'octet donnée en entrée sur le bus **data**,
- 2. il indique qu'il est occupé (signal **busy** à '1')
- 3. il envoie sur la ligne série **txd**, le bit de poids fort (7) de l'octet

puis, à chaque front montant de **clk**, il envoie les bits suivants de 6 à 0.

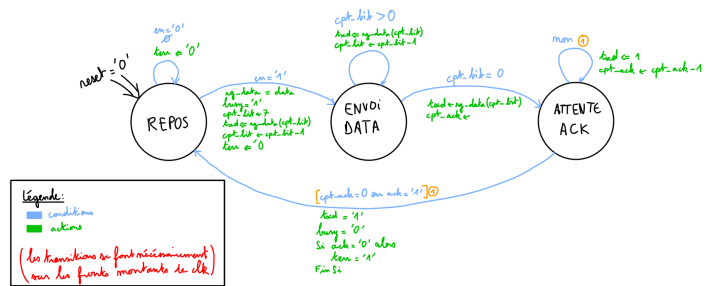
Une fois qu'il a envoyé les 8 bits, il remet la ligne série à '1' puis il attend, au plus 5 fronts montant de **clk**, la confirmation de la réception (signal **ack** actif à '1') pour se remettre au repos (**busy** à '0').

Si la confirmation de la réception n'arrive pas, il prévient le composant qui donne les ordres en positionnant un signal **terr** (erreur de transmission) à '1' pendant une période de l'horloge.

A tout moment, et de manière asynchrone, un **reset** (actif à '0') remet le composant au repos.

### Schéma

Antoine Rey le crack des schémas grâce à sa tablette de fou



```
when attente_ack =>
 busy <= '0';
 txd <= '1';
 if (cpt_ack > 0 and ack /= '1') then
 cpt_ack := cpt_ack - 1;
 else
 terr := not (ack = '1');
 etat <= repos;
 end if;
end case;
end if;

end process;

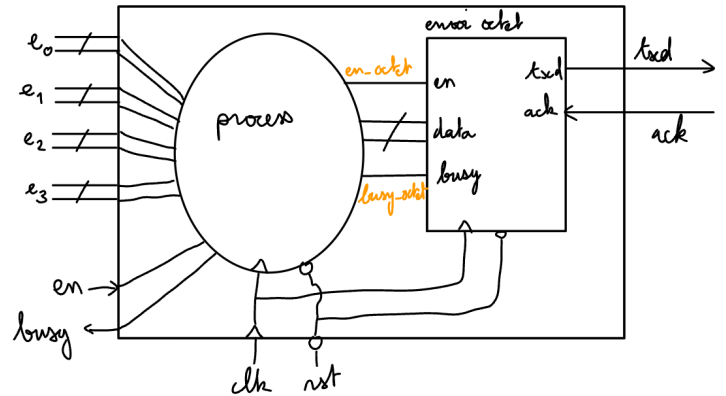
end behavioral;
```

## 2 Composant émettant 4 octets

On souhaite utiliser l'émetteur d'un octet dans un nouveau composant qui, lui, permettra d'émettre 4 octets à la suite [1] (<https://sanik.inpt.fr/pU-RySqURHSb6U5ALM87Lg?view#fn1>). Avant l'émission du premier octet, il y aura un temps d'attente de 5 tops d'horloge et entre chaque octet, un temps d'attente de 2 tops (quel est le moment où un octet est considéré comme envoyé ?).

[Question] : Développez un tel composant qui sera utile pour le mini-projet.

### Schémas



```
entity envoiOctet is
 port (
 clk, reset, en: in std_logic;
 data: in std_logic_vector(7 downto 0);
 txd, busy, terr: out std_logic;
);
end envoiOctet;

architecture behavioral of envoiOctet is

 type t_etat is (repos, Envoi_data, attente_ack);
 signal etat: t_etat;

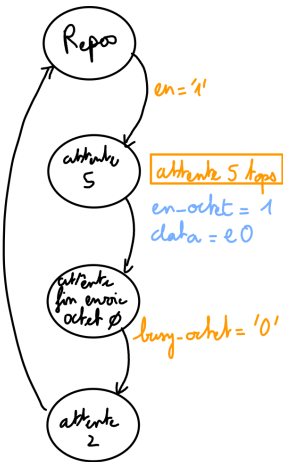
begin

 process(clk, reset)
 -- VARIABLES
 variable rg_data: std_logic_vector(7 downto 0);
 variable cpt_bit: natural;
 variable cpt_ack: natural;

 begin

 if (reset = '0') then
 -- RESET
 busy <= '0';
 terr <= '0';
 txd <= '1';
 etat <= repos;
 cpt_bit := 7;
 cpt_ack := 5;
 rg_data := (others => 'u');
 elsif (rising_edge(clk)) then
 -- MAIN
 case (etat) is
 when repos =>
 terr <= '0';
 if (en = '1') then
 rg_data := data;
 busy <= '1';
 cpt_bit <= 7;
 txd <= rg_data(cpt_bit);
 etat <= envoi_data;
 cpt_bit := cpt_bit - 1;
 end if;
 when envoi_data =>
 if (cpt_bit > 0) then
 txd <= rg_data(cpt_bit);
 cpt_bit := cpt_bit-1;
 -- inutile : etat <= envoi_data;
 else
 cpt_ack := 5;
 txd <= rg_data(cpt_bit);
 etat <= attente_ack;
 end if;
 end case;
 end if;
 end process;
```

### Code VHDL



1. on considérera qu'il n'y a guerre d'erreur de transmission et que le signal **terr** ne sera jamais actionné ← (https://sanik.inpt.fr/pU-RySqURHSb6U5ALM87Lg?view#fnref1)

```
entity envoi4octets is
 port (
 o1, o2, o3, o4: in std_logic_vector(7 downto 0);
 clk, reset, en : in std_logic;
 busy, txd : out std_logic;
);
end envoi4octets;

architecture ... of envoi4octets is

 component envoi0ctet
 port (
 clk, reset, en: in std_logic;
 data: in std_logic_vector(7 downto 0);
 txd, busy, terr: out std_logic;
);
 end component;

 type t_etat is (repos, attente5, envoi, attente2);
 signal etat : t_etat;
 signal busy_octet : std_logic;
 signal en_octet : std_logic;
 signal terr_octet : std_logic;
 signal data_octet : std_logic_vector(7 downto 0);

begin

 Serial: envoi0ctet port map(clk, reset, en_octet, data, txd, busy_octet, terr_o

 process(clk, reset)
 variable cpt_octet : natural; --indique quel octet on doit envoyer
 begin

 if(reset = '0') then
 -- RESET
 -- @TODO
 elsif (rising_edge(clk)) then
 -- MAIN
 case (etat)
 when repos =>

 when attente5 =>

 when attente2 =>

 when envoi =>

 end case;
 end if;
 end process;

end ...;
```

# Mini-Projet : Communication Carte Nexys 4 / Joystick PmodJSTK via protocole SPI

Sujet sur moodle : lien (<https://moodle-n7.inp-toulouse.fr/mod/resource/view.php?id=87731&redirect=1>)

## SPI

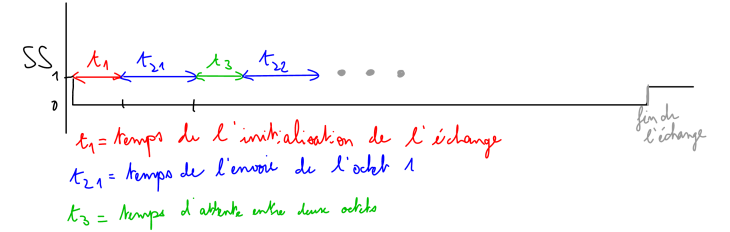
- protocole série bi-directionnel
- un 'maître', un ou plusieurs 'esclaves'
- 1 transmission = 1 trame = n octets

## Signaux:

- Slave select: **active low**: (1 = repos, 0=actif) -> échange de trame quand ss actif
- sclk : signal d'horloge, produit par le maître (fixe à 1 au repos (= hors transmission ou durant temps d'attente), varie entre 0 et 1 *uniquement* durant l'échange d'octet)
- mosi : master output, slave input
- miso : master input, slave output

L'échange d'une trame se déroule comme suit:

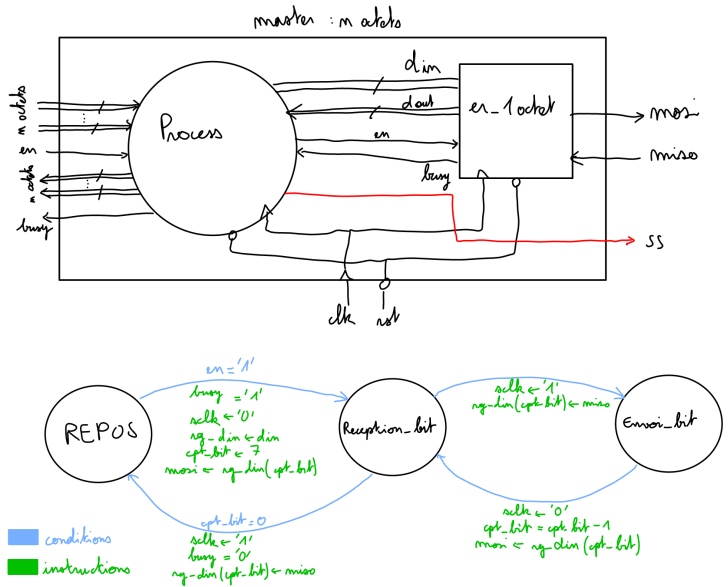
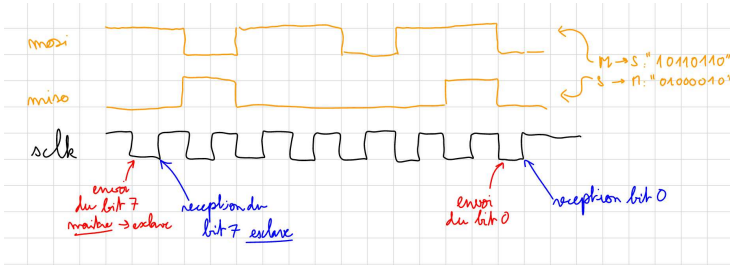
- t1 : temps d'attente après le passage de ss de 1 à 0 pour que l'esclave se prépare
- t2-1 : échange octet 1 (dans les deux sens)
- t3 : temps d'attente entre 2 octets
- t2-2 : deuxième échange d'octet
- t3
- ...
- t2n : après le dernier octet, ss remonte à 1 (*fin de la transmission*)



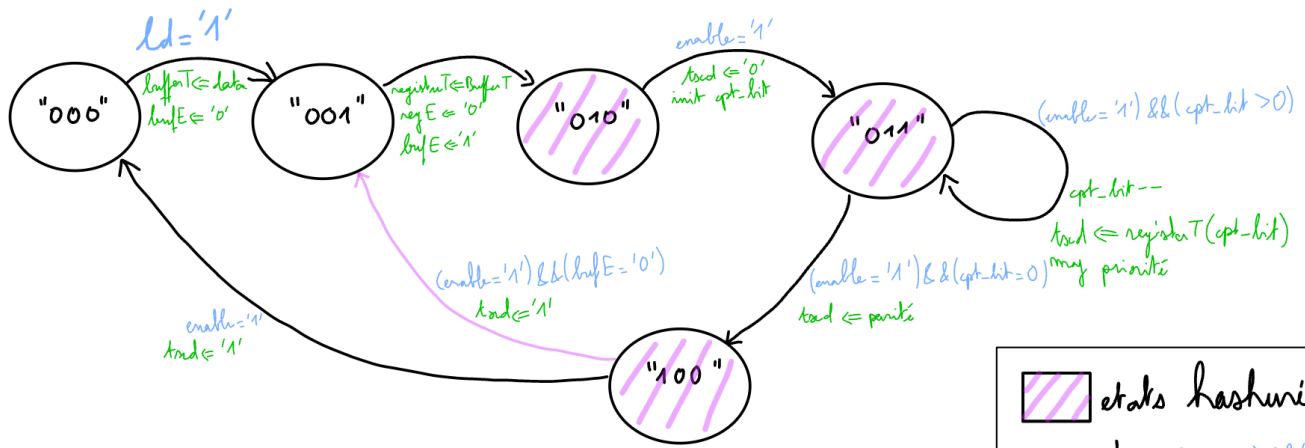
```
ss <- 0
initialisation de l'échange
répéter n fois
 - échange d'un octet
 - attente entre 2 octets
fin
ss <- 1
```


échange d'un octet :

- à chaque front descendant de sclk, on envoie un bit dans les deux sens
- à chaque front montant de sclk, on reçoit les bits émis lors du front descendant







 **etats hasheuris :**  
 on report:  $(ld = '1') \&\& (bufE = '1')$   
 $bufferT \leftarrow data$   
 $bufE \leftarrow '0'$