Rapport TP 2 - VHDL séquentiel

Introduction

Le but de ce TP est de découvrir le fonctionnement du VHDL Séquentiel. Contrairement au VHDL concurrentiel, le VHDL séquentiel fait intervenir des « process », définis par une liste de constructions introduites par le mot Process. Lors d'une modélisation, ces instructions se déroulent séquentiellement, au contraire du comportement concurrentiel pour lequel les instructions s'exécutent en parallèle, sans tenir compte de l'état passé des signaux.

Exercice 1 – Compteur synchrone 2 bits avec Reset asynchrone

Pour réaliser ce compteur nous disposons :

- D'un bouton poussoir faisant office de signal d'horloge : BTN_D
- D'un bouton Reset : BTN_G
- D'un signal de sortie permettant d'afficher le compteur : LED_1_0, affiché sur 2 LED

Q1.

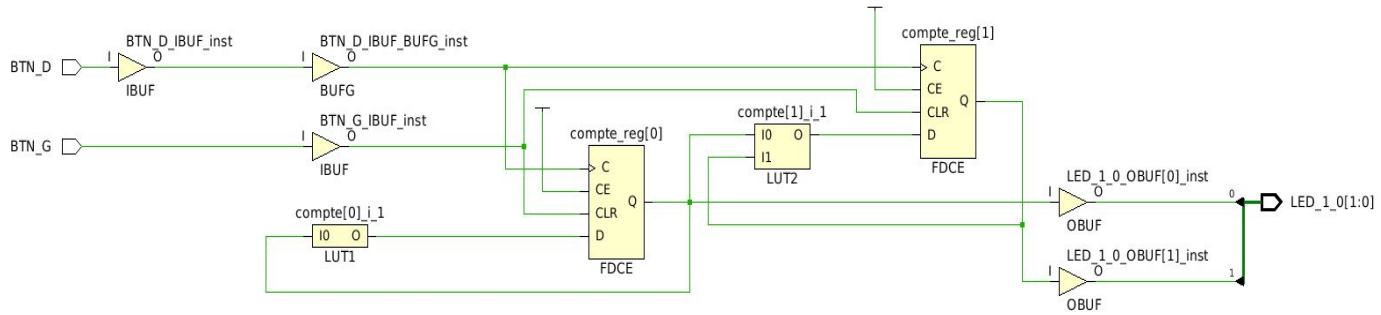
```
entity ex01 is
        port (BTN_D, BTN_G: in bit;
        LED_1_0: out integer range 0 to 3)
end ex01;

architecture Behavioral of ex01 is
begin
        process (BTN_D, BTN_G)
        variable compte: integer range 0 to 3;
        begin
        if BTN_D'event AND BTN_D = '1' AND BTN_G='0' THEN
                compte:= compte + 1;
        end if;
        IF BTN_G'event and BTN_G = '1' THEN
                compte := 0;
        end if;
        LED_1_0 <= compte;
        end process;
end Behavioral;
```

Le signal de sortie LED_1_0 apparaît sur un groupe de 2 leds, il est donc codé sur 2 bits. Il peut ainsi prendre 4 valeurs : 00, 01, 10, 11 (en décimal 0, 1, 2, 3). Ces valeurs nous indiquent la valeur du compteur (le nombre d'appuis sur BTN_D avant Reset).
Dans le VHDL nous avons donné à LED_1_0 le type *integer* allant de 0 à 3.
A l'intérieur du Process, nous avons déclaré une nouvelle variable « compte ». Cette variable est de même type que LED_1_0. Cette variable fait office de compteur : elle garde en mémoire le nombre d'appuis sur BTN_D. Ce nombre correspond également aux différents états du système (0, 1, 2 ou 3).
A la fin du Process, LED_1_0 prend la valeur de *compte*. On utilise l'intermédiaire de *compte* pour connaître la valeur de LED_1_0 car un signal ne peut pas être testé.

Q2.
Pour la partie simulation, nous avons forcé le front montant de l'horloge (BTN_D) à une période de 1 ms et l'activation de BTN_G (Reset) à 5 ms. Nous avons lancé enfin la simulation: le système va remettre à 0 le compteur après une période de 5 ms, ou quand le compteur dépasse 3 (après 3 ms).

Remarque : Dans le VHDL indiqué Q.1, nous n'avons mis d'instructions concernant le cas où, si compteur=3 et qu'on appuie sur BTN_D, le compteur doit être remis à 0 car *compte* ne peut pas être incrémenté au-delà de la valeur 3.
En effet, lors de l'implémentation et du « vrai » fonctionnement du programme, cela est géré automatiquement. En simulation cependant, il faut l'ajouter, sinon le compteur augmente au-delà de 3 tant qu'on appuie pas sur Reset, ce qui n'a pas de sens car LED_1_0 ne peut pas prendre de valeur au-delà de 3.

Q3.



Composants: Dans ce schéma, le compteur a 2 bits qui sont gérés par 2 boîtes FSM_sequential_comtpeur_reg (FDCE) avec la porte CE toujours =1, à partir de lequel on peut comprendre que la sortie Q toujours prends la valeur de l'entrée D pendant la transition de l'horloge. Porte CLR est connecté directement avec l'entrée BTN_G et asynchrone. L'entrée de bascule est générée par la boîte FSM_sequentiel_compteur_i_1 (LUT)

Fonctionnement du système: À chaque front montant d'horloge (BTN_D), la sortie Q de chaque bascule va récupérer la valeur de D pour retransmettre à sortie LED_1_0_OBUF_inst / à prochain bascule D. Pour LUT1 (FSM_sequential_compteur[0]_i_1) ou le poids faible du compteur, il a une entrée connecté avec la sortie Q et une sortie connecté avec l'entrée D de bascule FSM_sequential_comtpeur_reg[0], on peut voir que la sortie est l'inverse de l'entrée. Pour le poids fort du compteur, c'est presque même fonctionnement de poids faible (avec FDCE et LUT2) mais LUT2 a 2 entrées, l'un connecte avec la sortie Q de FDCE 1 (entrée 1) et l'autre connecte avec la sortie Q de FDCE 2 (entrée 2), LUT2 = entrée 1 XOR entrée 2

Q4.
Le circuit fonctionne comme prévu. Quand on presse le bouton BTN_D, on a fait un front montant pour l'horloge, et si le bouton BTN_G n'est pas pressé le compteur va incrémenter sa valeur. Au cas où BTN_G serait actif, le compteur va se remettre à 0 immédiatement sans tenir compte du front d'horloge => Compteur synchrone avec reset asynchrone.

Exercice 2 - Compteur synchrone 2 bits avec Reset synchrone

Q1.

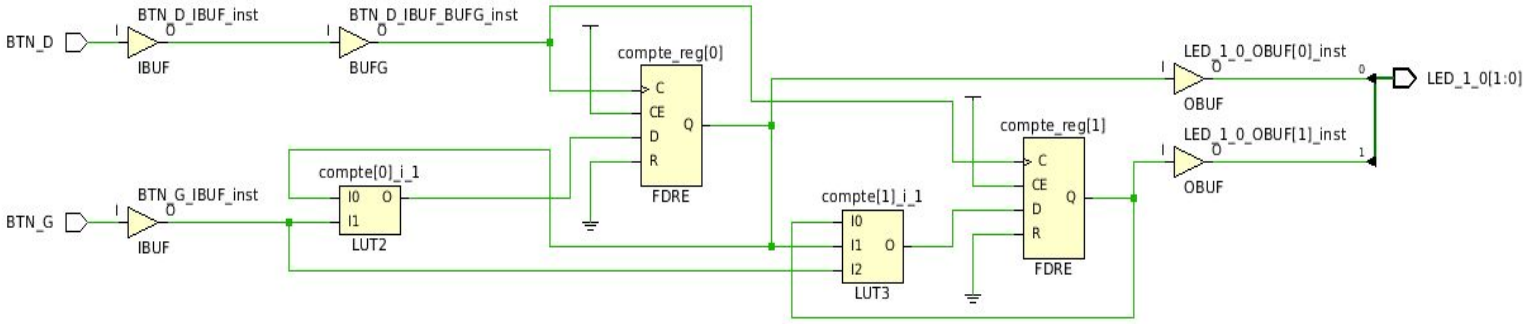
```
entity compt_sync is
        port(BTN_D, BTN_G: in bit;
        LED_1_0: out integer range 0 to 3);
end compt_sync;

architecture Behavioral of compt_sync is
begin
   process (BTN_D)
   variable compte: integer range 0 to 3;
   begin
        if (BTN_D'event AND BTN_D='1') then
                if BTN_G = '0' then compte:= compte+1;
                else compte:=0;
        end if;
      end if;
   LED_1_0 <= compte;
   end process;
end Behavioral;
```

Pour réaliser ce compteur nous disposons exactement des mêmes éléments que pour l'exercice précédent. La différence est que cette fois le Reset est synchrone : à chaque signal d'horloge (ici, appui sur le BTN_D), le système va vérifier que BTN_G n'est pas actif (qu'il est donc égal à 0). Le seul cas ou le Reset sera pris en compte sera donc au moment d'un signal d'horloge. Autrement dit, pour remettre le compteur à 0, il faut appuyer à la fois sur BTN_G et sur BTN_D.

Ce n'était pas le cas dans l'exercice 1, ou un appui sur BTN_G à n'importe quel moment (donc par forcément simultanément à l'appui sur BTN_D) permettait de remettre le compteur à 0.

Q2.
A la simulation, nous pouvons constater que la valeur du compteur est remise à 0 quand le compteur dépasse 3 ou BTN_D et BTN_G = 1.

Q3.



En comparant avec le schéma de l'exercice précédent, on voit que la bascule a une nouvelle entrée R, toujours égale à 0, pour montrer que le reset est synchrone. Les LUT ont une nouvelle entrée, BTN_G, pour voir s'il y a un signal de reset. En bref, le reset dans l'exercice 1 va remettre directement la valeur de Q = 0, tandis que dans l'exercice 2 Q ne dépend plus de reset. Reset remet en 0 la valeur de LUT (compteur) qui change la valeur seulement lors d'un front montant de l'horloge.

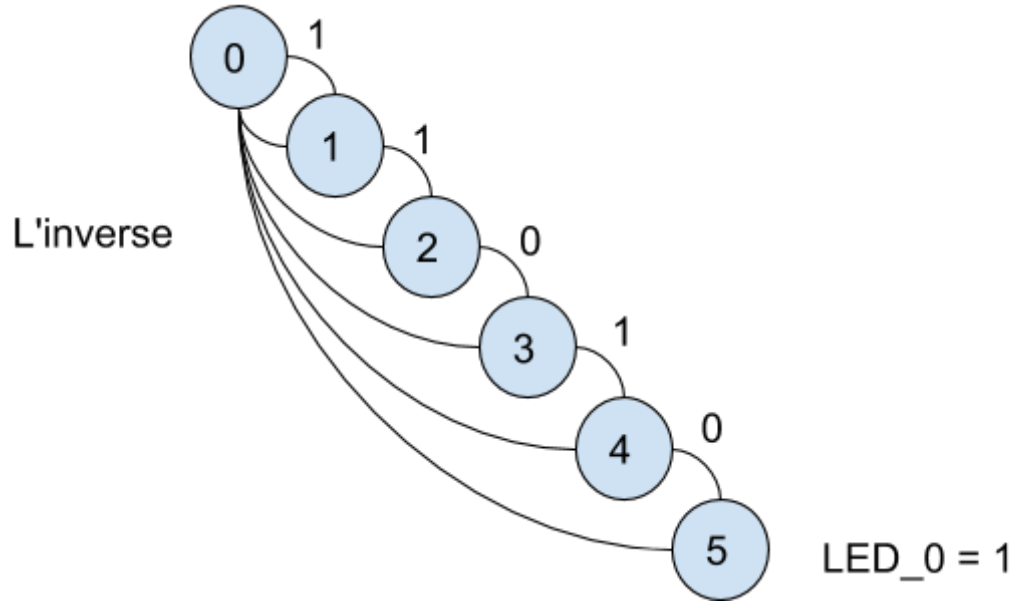
Q4.
Le circuit fonctionne comme prévu. Quand on presse en même temps BTN_D et BTN_G, le compteur se remet à 0. Si on presse seulement BTN_G, le compteur ne change pas de valeur; si on presse le bouton BTN_D, le compteur va s'incrémenter si BTN_G=0.

Exercice 3 - Détecteur de code en transmission 1 bit - Fausse entrée oblige à recommencer le code

Pour réaliser ce compteur nous disposons :

- D'un bouton poussoir faisant office de signal d'horloge : BTN_D
- D'un interrupteur: SW_0
- D'un signal de sortie permettant d'afficher l'alarme : LED_0
- D'un signal de sortie permettant d'afficher l'état: LED_7_4, affiché sur 4 bits mais réellement 3 bits (5 états)

Q1.



Q2.

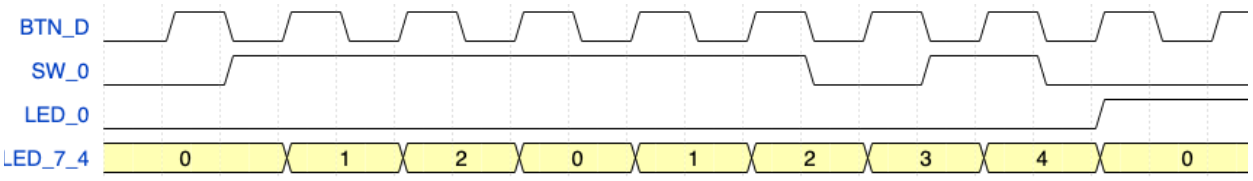
```
entity ex3 is
   port (SW_0, BTN_D: in bit;
         LED_0: out bit;
         LED_7_4: out integer range 0 to 5);
end ex3;
```

```
architecture Behavioral of ex3 is
begin
   process (BTN_D)
   variable state: integer range 0 to 5;
   begin
      if (BTN_D'event AND BTN_D = '1') then
         case state is
            when 0 => if SW_0 = '1' then state :=1; end if;
            when 1 => if SW_0 = '1' then state :=2; else state :=0; end if;
            when 2 => if SW_0 = '0' then state :=3; else state :=0; end if;
            when 3 => if SW_0 = '1' then state :=4; else state :=0; end if;
            when 4 => if SW_0 = '0' then state :=5; else state :=0; end if;
            when others => state :=0;
         end case;
      end if;
      if state = 5 then
         LED_0 <= '1';
      else
         LED_0 <= '0';
      end if;
      LED_7_4 <= state;
   end process;
end Behavioral;
```

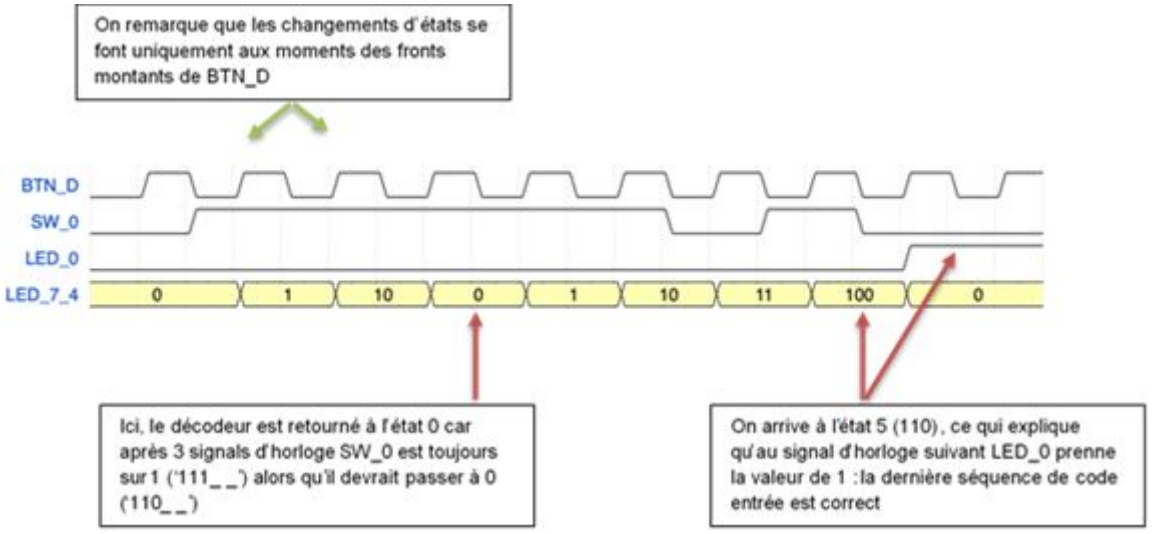
Q3.

Nous avons oublié de faire la capture d'écran de la simulation du détecteur mais nous avons compris son fonctionnement. À chaque état, si on est déjà plus en 0 (1 à 4), si on détecte une valeur erronée (par exemple '0' à l'état 1 alors qu'on attend un '1' pour passer à l'état 2), on retourne tout de suite à l'état 0 pour recommencer tout le code. Le changement d'état s'effectue seulement lors de front montant de l'horloge (lorsqu'on appuie sur BTN_D). La sortie LED_0 = 1 si et seulement si SW_0 = "11010".

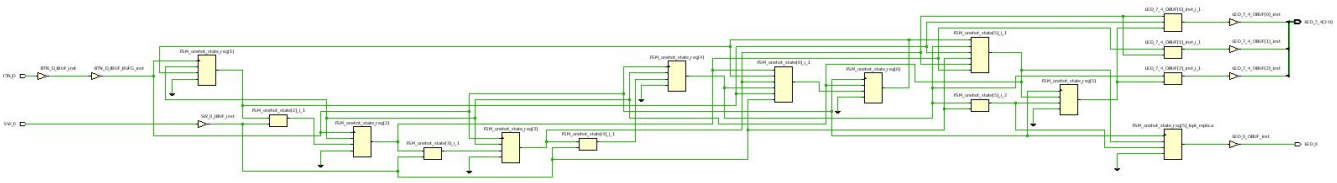
Pour simuler le détecteur, nous décidons d'ajouter un dessin manuel représentant une simulation:



Interprétation:



Q4.



Dans le FPGA, LED_7_4 est une sortie de 4 bits (15 états maximum) mais dans le code VHDL, on utilise seulement 5 états, donc on a besoin de seulement 3 bits pour représenter l'état du détecteur. C'est pourquoi dans le schéma, LED_7_4[3] n'est pas représenté, car dans le code VHDL (toujours mise à 0 dans ce cas là), nous avons mis output LED_7_4 un signal type integer range 0 à 5. À partir du schéma et les portes sur FPGA, nous trouvons que ce n'est pas cohérent, nous proposons alors l'utilisation LED_6_4 comme l'affichage d'états au lieu de LED_7_4.

Q5.

Nous ne disposons pas du schéma RTL pour le détecteur. Donc nous ne pouvons pas trouver les équations logiques des bits d'états utilisées pour générer la sortie.

Les bits d'états sont représentés en 6 bits (6 sorties de S0 à S5) et les équations logiques ont de paramètres: les bits d'états, SW_0 et aussi les LED_7_4.

Q6.

Nous avons constaté que le circuit fonctionnait comme prévu. On change la valeur de SW_0 (interrupteur à 1 ou 0 selon l'état et selon le respect ou non du code correct à décoder) avant de presser le bouton BTN_D pour transmettre cette valeur. Au cas-où on trouve une fausse entrée, l'état va être remis à 0, c'est-à-dire le LED_7_4 = 0. Si le code 11010 est détecté, la sortie LED_0 = 1 (alarme). La sortie LED_7_4 indique l'état correspondant à chaque front montant d'horloge, dans ce cas-là, en 3 bits au lieu de 4 bits car LED_7_4[3] est toujours égale à 0.

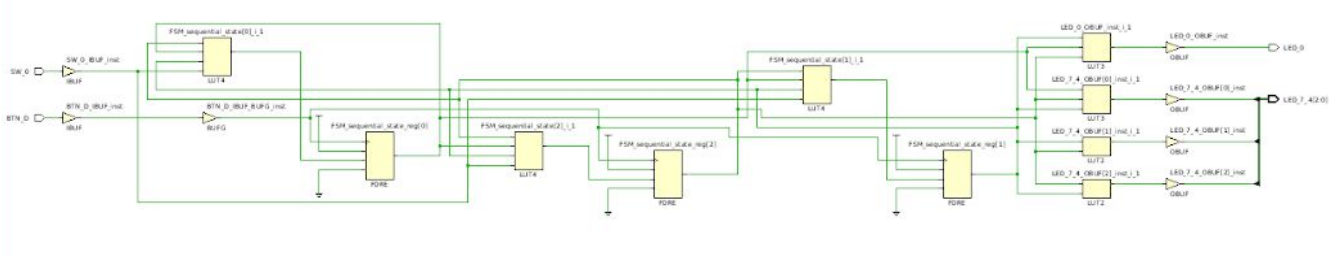
Exercice 4 - Détecteur de code en transmission 1 bit - Fausses entrées négligées

Q1. Code VHDL avec fausses entrées négligées

```
entity exo4 is
PORT(SW_0, BTN_D : IN BIT;
        LED_0 : OUT BIT;
        LED_7_4 : out integer range 0 to 5) ;
end exo4;
architecture Behavioral of exo4 is
begin
process(BTN_D)
variable state : integer range 0 to 5;
begin
if (BTN_D'Event and BTN_D = '1') then
case state is
        when 0 => if (SW_0 = '1') then state := 1; end if ;
        when 1 => if (SW_0 = '1') then state := 2; end if ;
        when 2 => if (SW_0 = '0') then state := 3; end if ;
        when 3 => if (SW_0 = '1') then state := 4; end if ;
        when 4 => if (SW_0 = '0') then state := 5; end if ;
        when others => state := 0;
end case ;
LED_7_4 <= state;
end if ;
if (state = 5) then LED_0 <= '1';
else LED_0 <= '0';
end if;
```

```
end process ;
end Behavioral;
```

Q2.



Bien qu'on ait le même nombre de bits d'états (6 bits), les équations de sorties sont plus compliquées et font intervenir beaucoup plus de signaux.