Leonardo D'ALESSIO - GI01
Soulaymane KEBLI - GI01

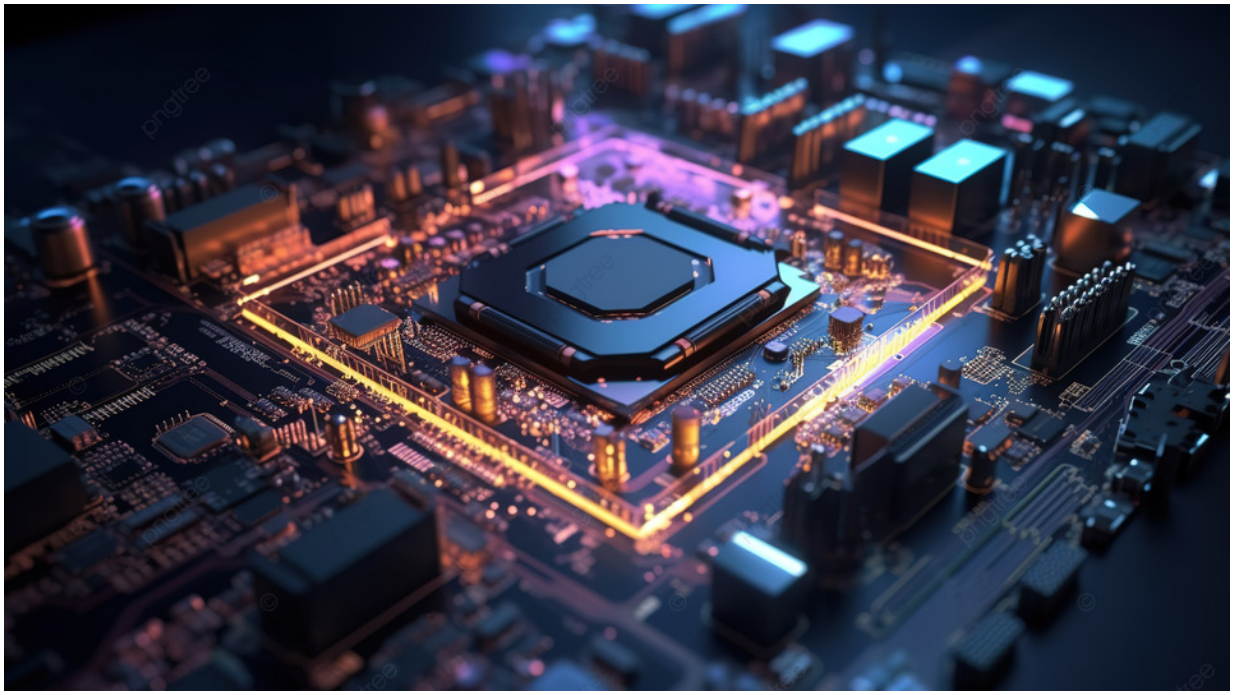
Université de Technologie de Compiègne

MI01 - Structure d'un calculateur

Responsable
Mr. Shawky Marc

TP2 - VHDL SEQUENTIEL

Responsable TP
Mr. Sanahuja Guillaume



TP1 - Groupe n°6

Sommaire

Table des figures

1 Exercice 1 - Compteur synchrone 2 bits avec reset asynchrone

1.1 Objectif

Dans cet exercice, nous devons modéliser un compteur synchrone 2 bits avec reset asynchrone. Ce compteur doit s'incrémenter à chaque appui sur le bouton poussoir de droite qui fait office de signal d'horloge.
Nous devons utilisé pour cela :

- BTN_D ; la variable d'entrée qui caractérise le bouton de droite et l'horloge asynchrone active sur front montant,
- BTN_G ; la variable d'entrée qui caractérise le bouton de gauche et le reset actif au niveau haut (BTN_G = '1'),
- LED_1_0 ; groupe des deux leds servant à caractériser la valeur du compteur.

1.2 Modélisation du compteur en VHDL

Nous avons écris le programme ci-dessous dans le fichier cpta.vhd :

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

architecture Behavioral of cpta is
    begin
    process(BTN_G, BTN_D)
        variable i : integer range 0 to 3 := 0;
        begin
            if (BTN_G ='1') then
                i:=0;
            elsif (BTN_D'event and BTN_D = '1') then
                i := i + 1;
            end if;
            LED_1_0 <= i;
    end process;
end Behavioral;
```

FIGURE 1 – *Programme VHDL Compteur synchrone avec reset asynchrone*

Pour réaliser un compteur synchrone 2 bits avec reset asynchrone en VHDL, on a utilisé un process qui agit sur le bouton gauche(le reset) et sur le bouton droit (l'horloge). On teste ensuite avec comme ordre de priorité, le reset en vérifiant s'il égale à 1. Si oui, on réinitialiser le compteur i à 0. On teste ensuite s'il y a un événement sur l'horloge, c'est à dire si on a pressé le bouton droit. Si oui, le compteur i est incrémenté de 1 modulo 3. Enfin, on affecte le compteur i à LED_1_0.

1.3 Simulation du circuit et observation

Une fois l'écriture du programme terminée, on a lancé sa simulation. Pour cela, on a forcé tout d'abord l'horloge (BTN_D) à 1 avec une période de 10µs et on a laissé le reset (BTN_G) à 0.

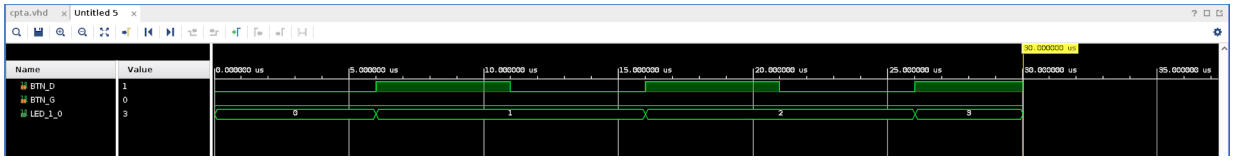


FIGURE 2 – *Simulation compteur synchrone avec l'horloge à 1 et le reset à 0*

On peut bien voir les valeurs de LED_1_0 allant de 0 à 3 en observant les fronts montants et descendants. On affecte maintenant à l'horloge la valeur 0 et au reset la valeur 1 afin de réinitialiser le compteur :

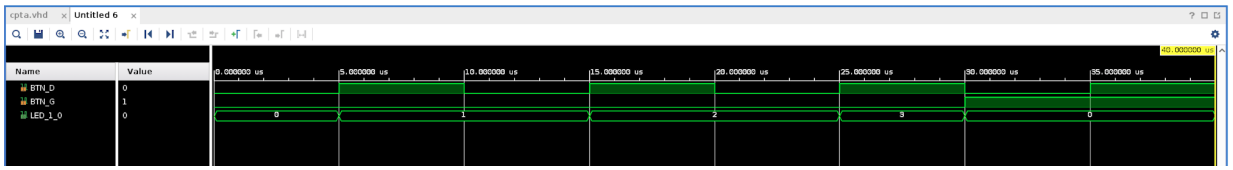


FIGURE 3 – *Simulation compteur synchrone avec l'horloge à 0 et le reset à 1*

On peut observer que le compteur se réinitialise correctement. De plus, étant un compteur synchrone avec un reset asynchrone on peut s'apercevoir que le front montant du reset s'active sur un front descendant de l'horloge et réinitialise directement le compteur sans attendre.

1.4 Implémentation et schéma RTL

Après avoir lancé l'implémentation, on obtient le schéma RTL suivant :

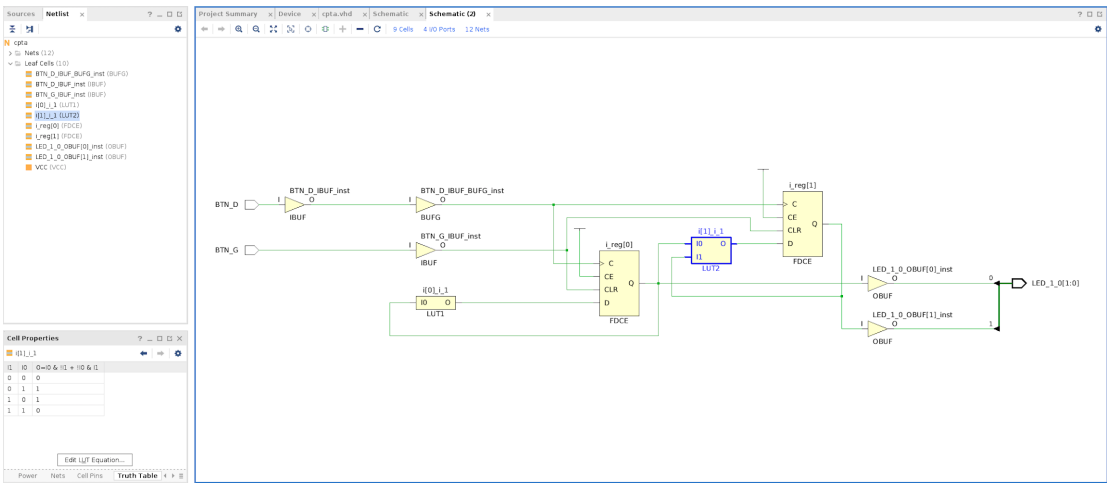


FIGURE 4 – *Schéma RTL compteur synchrone avec reset asynchrone*

On peut voir que le logiciel à modéliser notre compteur à l'aide principalement de bascule D (bloc FDCE) et d'inverseur. Ces bascules vont servir à stocker la valeur du compteur et à le réinitialiser sur front montant du reset car on peut voir que le BTN_G est branché au CLR des bascules. les inverseurs vont permettre notamment de générer un "NOT_RESET" à partir du signal de réinitialisation du reset asynchrone. Enfin, les composant LUT vont permettre de mettre à jour les bascules D du compteur à chaque front montant de l'horloge en fonction de la table de vérité (ici elle correspond à celle d'un ou exclusif).

Dans ce schéma, on peut voir que le BTN_G (reset) est bien relié au CLR (clear) de chaque FDCE. Le BTN_D (horloge), lui, est bien relié au C (clock) de chaque FDCE. On a ensuite deux sorties, ce sont les LED qui représentent l'état du compteur. Le compteur est codé sur 2 bits, donc il y a une LED pour le bit de poids faible et une pour le bit de poids fort. Chaque bit a son FDCE et sa Look Up Table : "etat_reg[0]" et LUT1 pour le bit de poids faible et "etat_reg[1]" et LUT2 pour le bit de poids fort. Lorsqu'on cherche à implémenter un compteur, il est nécessaire de se souvenir de l'état précédent, en effet chaque nouvel état est égal à l'état précédent plus 1. C'est pour cela que dans chaque FDCE, la sortie Q est reliée à l'entrée D. Plus précisément, la sortie Q va donner sa valeur à la LED pour l'état courant et en même temps à une LUT qui va se charger de modifier cette valeur (voir paragraphe suivant sur le fonctionnement des LUT). Une fois modifiée, cette valeur est renvoyée au FDCE et stockée dans D. C'est la valeur de l'état suivant, mais Q possède encore la valeur de l'état courant. Ce n'est qu'au prochain front montant d'horloge, que Q prendra la valeur de D, et qu'on sera passés à un nouvel état.

Intéressons nous maintenant à la différence entre les deux LUT. La LUT1 correspond au non logique, elle transforme tous les 0 en 1 et tous les 1 en 0. Or, quand on incrémente un nombre binaire de 0 à 3, on observe que le bit de poids faible suit le schéma 0->1->0->1. Une incrémentation de 1 correspond donc bien à un non logique. La LUT2 prend une entrée supplémentaire, c'est le résultat de la LUT1. En effet, lors d'une addition, la valeur du bit de poids fort dépend de celle du bit de poids faible.

2 Exercice 2 - Compteur synchrone 2 bits avec reset synchrone

2.1 Objectif

Dans cet exercice, nous devons toujours modéliser un compteur synchrone 2 bits mais cette fois-ci avec un reset synchrone. Ce compteur doit s'incrémenter à chaque appui sur le bouton poussoir de droite qui fait office de signal d'horloge et se réinitialiser à chaque appui sur le bouton poussoir gauche.

Pour cela nous gardons les mêmes variables d'entrées et de sorties que l'exercice précédent.

2.2 Modélisation du compteur en VHDL

Nous avons écris le programme ci-dessous dans le fichier cpts.vhd :

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

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

FIGURE 5 – *Programme VHDL Compteur synchrone avec reset synchrone*

Le programme VHDL reste assez similaire que celui de l'exercice précédent à l'exception de quelques différences. En effet, on change les paramètres du process ; cette fois-ci le process n'agira que sur le bouton droit (l'horloge). De plus, on effectue quelques changement au niveau des testes de priorités. on teste d'abord s'il y'a un événement sur le bouton droit puis on teste si le reset est activé. Ces modifications permettent de rendre le reset du compteur synchrone.

2.3 Simulation du circuit et observation

Une fois l'écriture du programme terminée, on a lancé sa simulation. Tout comme pour l'exercice d'avant, on commence forcé l'horloge (BTN_D) à 1 avec une période de 10µs et on a laissé le reset (BTN_G) à 0.

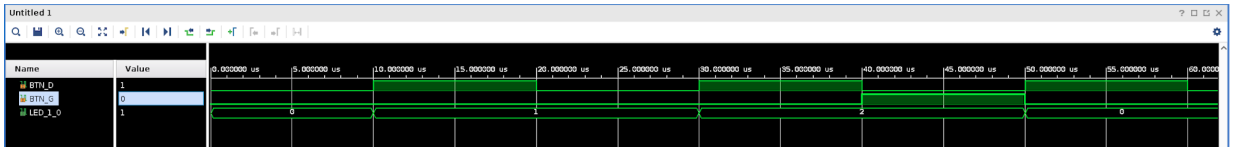


FIGURE 6 – *Simulation compteur synchrone avec reset synchrone*

On peut bien voir le compteur synchrone avec les différentes valeurs croissante 0,1,2. On a forcé le reset au milieu de la valeur 2 et on peut s'apercevoir cette fois-ci que le reset s'effectuera au prochain front montant de l'horloge.

Sur notre simulation cela peut paraître difficile à voir en raison d'un décalage de période. En effet l'horloge était réglée sur une période de 20µs et le reset avec un pas de 10µs.

2.4 Implémentation et schéma RTL

Après avoir lancé l'inplémentation, on obtient le schéma RTL suivant :

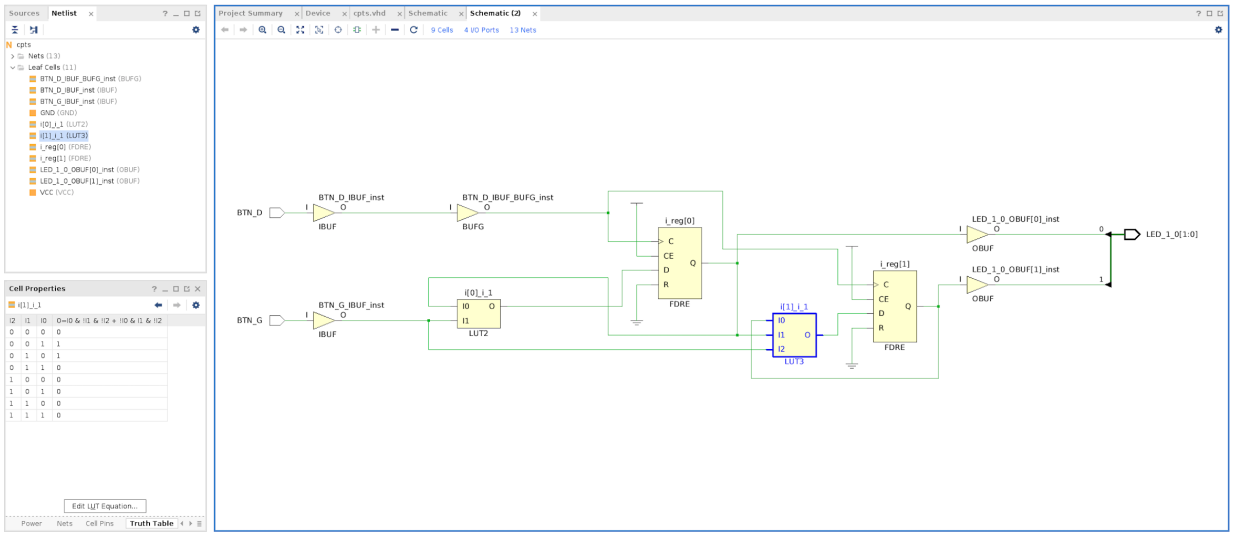


FIGURE 7 – *Schéma RTL compteur synchrone avec reset synchrone*

Dans l'exercice précédent, le reset était asynchrone car le BTN_G était relié directement à l'entrée CLR du FDCE, qui lorsqu'elle était à 1, mettait Q à 0 peu importe la valeur de la clock C.

Ici, le BTN_G est une entrée supplémentaire pour les deux LUT. Le reste du fonctionnement est le même. Si on regarde les 2 LUT, on s'aperçoit que ce sont les mêmes que pour l'exercice précédent, à l'exception de la première colonne qui correspond au BTN_G. Si BTN_G est à 0, les résultats sont les mêmes que pour l'exercice précédent. Mais s'il est à 1, la sortie sera forcément à 0, peu importe les autres entrées. C'est donc parce que BTN_G est dans les LUT que le reset est synchrone. En effet, si BTN_G est à 1, D prendra la valeur 0. Mais ce n'est qu'au prochain front montant d'horloge que Q prendra la valeur de D et donc que les diodes s'éteindront pour modéliser la réinitialisation du compteur.

3 Exercice 3 - Détecteur de code

3.1 Objectif

L'objectif de l'exercice 3 est de réaliser un détecteur de code spécial sur une ligne de transmission série (de 1 bit). Dès que le code 11010 est détecté, le système le signale en mettant une ligne d'alarme à 1. L'alarme doit rester inactive pour toute autre combinaison de codes. Dans un premier temps, une entrée erronée obligera à recommencer tout le code. Les changements de niveau de l'entrée série sont synchronisés à l'horloge. Nous utiliserons les signaux suivants dans notre VHDL :

— SW_0, un interrupteur prenant la valeur du bit reçu sur la ligne série

— BTN_D, un bouton poussoir faisant office de signal d'horloge

— LED_0, une LED permettant d'indiquer si le code vient d'être entré (LED_0 = 1) ou non (LED_0 = 0).

3.2 Modélisation sous forme de machine à états

Nous avons tout d'abord conçu cette machine à états :

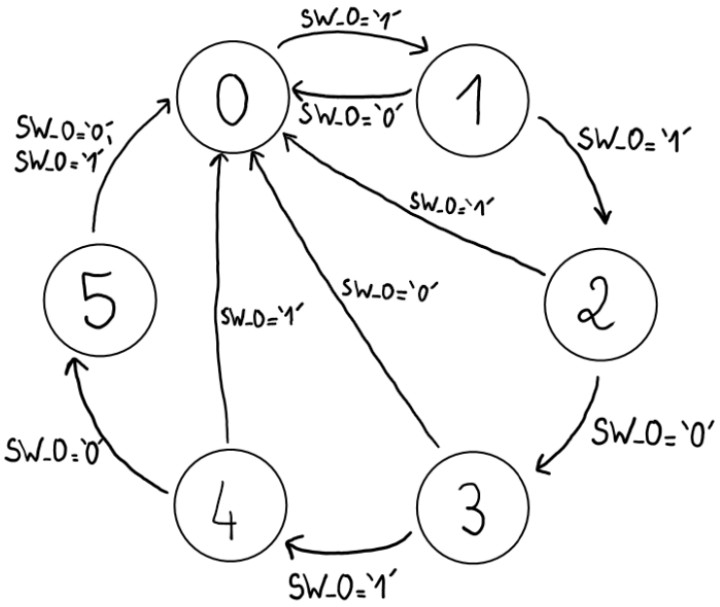


FIGURE 8 – *Machine à états détecteur de code*

3.3 Modélisation du détecteur de code en VHDL

Nous avons écris le code suivant dans le fichier detecteur.vhd :

```
entity detecteur is
    port(SW_0, BTN_D : in STD_LOGIC;
    LED_0 : out STD_LOGIC);
end detecteur;

architecture Behavioral of detecteur is
    begin
    process (BTN_D)
        variable etat : integer range 0 to 6 := 0;
            begin
            if (BTN_D'event and BTN_D = '1') then
                case etat is
                    when 0 => if (SW_0 = '1') then etat := 1; end if;
                    when 1 => if (SW_0 = '1') then etat := 2; else etat := 0 ; end if;
                    when 2 => if (SW_0 = '0') then etat := 3; else etat := 0 ; end if;
                    when 3 => if (SW_0 = '1') then etat := 4; else etat := 0 ; end if;
                    when 4 => if (SW_0 = '0') then etat := 5; else etat := 0 ; end if;
                    when others => etat := 0;
                end case;
                if (etat = 5) then
                    LED_0 <= '1';
                else
                    LED_0 <= '0';
                end if;
            end if;
    end process;
end Behavioral;
```

FIGURE 9 – *Programme VHDL détecteur de code*

3.4 Simulation du circuit et observation

Voici le chronogramme de la simulation que l'on a obtenu après avoir entré le code "11010" :

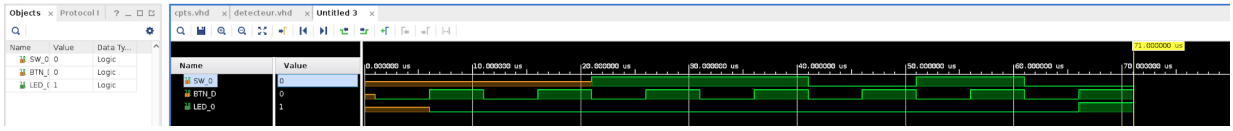


FIGURE 10 – *Simulation détecteur de code*

On observe que, à chaque fois que BTN_D est active (ce qui est modélisé avec un front montant), le détecteur lit l'information présente sur SW_0 (1 ou 0). Ainsi, on voit que , à partir de 20 us, on fait lire le code "11010" au détecteur, et que dès que le dernier 0 est inséré (66 us), LED 0 bascule en 1, ce qui signifie dans la vraie vie qu'elle serait allumée et se qui se traduit par un code valide.

3.5 Implémentation et schéma RTL

Après avoir lancé l'inplémentation, on obtient le schéma RTL suivant :

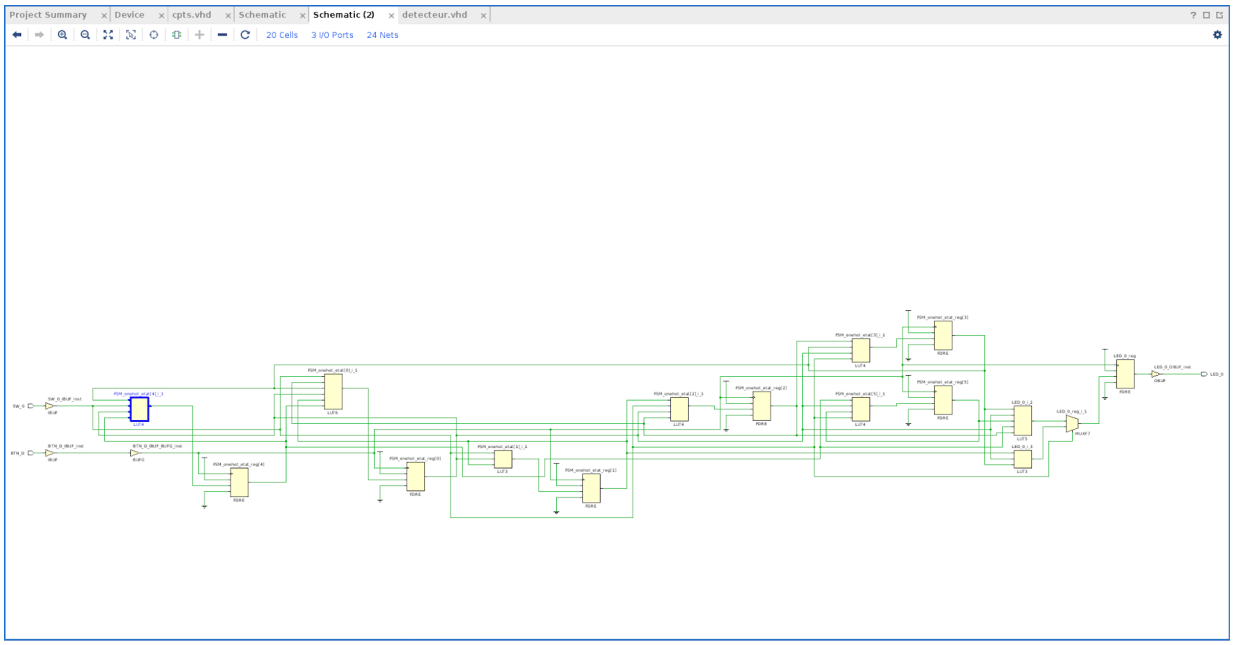


FIGURE 11 – *Schéma RTL détecteur de code*

On observe directement que Vivado/Xilinx a implémenté le détecteur en utilisant la méthode One-hot, car on voit la présence de "FSM_onehot_etat_reg[n]" et "FSM_onehot_etat[n]". Ainsi, au lieu d'utiliser des bits pour représenter les états, on utilise directement 5 registres pour représenter les états, ce qui fait qu'on ne peut pas déterminer le nombre de bits utilisés. Le schéma est donc cohérent, parce qu'on a bien besoin de 5 états (un pour chaque chiffre du code).

3.6 Équation logique

D'après le fichier "Remarques TP 2 VHDL - exercice 3", "Quant aux équations logiques, il n'est pas nécessaire de les fournir, savoir simplement qu'elles seront sous la forme y(t)=F(x(t),x(t-1), x(t-2),...), en rebouclant la sortie de certains registres d'état vers les entrées d'autres registres."

3.7 Programmation du FPGA

Nous n'avons malheureusement pas eu le temps de prendre des photos pour le fonctionnement du détecteur sur la carte. Voici ce que l'on a observé :

On a tout d'abord essayé de directement mettre le code "11010" afin de voir si la LED 0 s'allumait. Ainsi, on a procédé de la manière suivante : on changeait d'abord SW_0 dans la position voulue (haut pour 1, bas pour 0), puis on appuyait brièvement sur BTN_D pour que le système change d'état. Ainsi, après avoir inséré le dernier 0, la LED 0 s'est allumée, ce qui montrait que la programmation avait bien fonctionné.

Ensuite, pour voir si la machine à état se réinitialise après une mauvaise entrée, on a réalisé le test suivant : on a inséré "110", puis "0" (erreur) , puis "10" (faire comme si l'erreur ne s'était pas passée). On a dès lors observé que la LED 0 ne s'était pas allumée, ce qui montre que la machine s'était bien réinitialisée à l'état 0.

4 Exercice 4 - Détecteur de code avec fausses entrées négligées

4.1 Objectif

Les objectifs restent les mêmes que l'exercice précédent à l'exception des fausses entrées qui seront négligées. En effet, pour l'exercice 3 une entrée erronée obligeait à recommencer tout le code. Dans l'exercice 4, celle-ci sera négligée.

4.2 Modélisation sous forme de machine à états

Nous avons tout d'abord conçu cette machine à états :

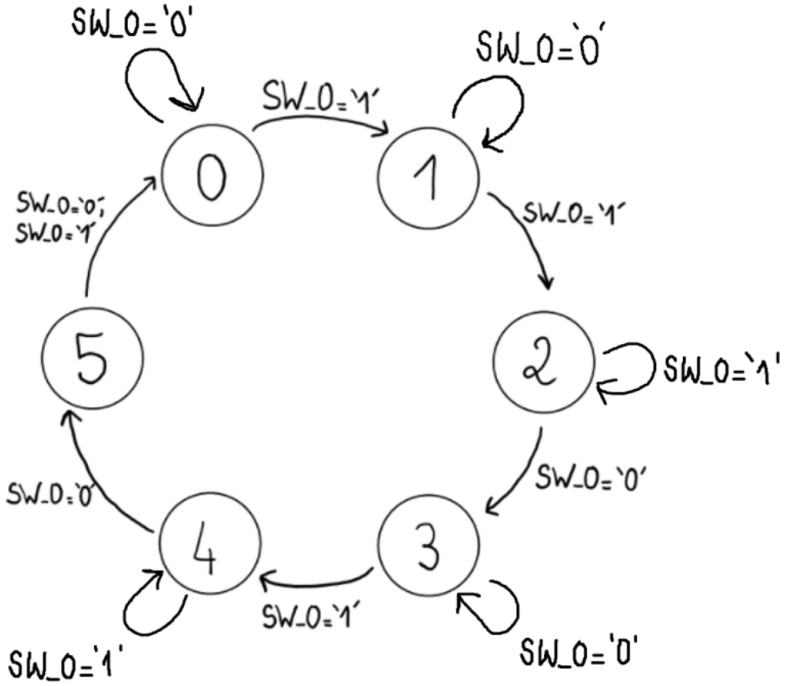


FIGURE 12 – *Machine à état détecteur de code avec fausses entrées négligées*

4.3 Modélisation du détecteur de code en VHDL

Nous avons écris le code suivant dans le fichier detecteur2.vhd :

```
entity detecteur2 is
    port(SW_0, BTN_D : in STD_LOGIC;
    LED_0 : out STD_LOGIC);
end detecteur2;

architecture Behavioral of detecteur2 is
    begin
    process (BTN_D)
        variable etat : integer range 0 to 6 := 0;
            begin
                if (BTN_D'event and BTN_D = '1') then
                    case etat is
                        when 0 => if (SW_0 = '1') then etat := 1; end if;
                        when 1 => if (SW_0 = '1') then etat := 2; end if;
                        when 2 => if (SW_0 = '0') then etat := 3; end if;
                        when 3 => if (SW_0 = '1') then etat := 4; end if;
                        when 4 => if (SW_0 = '0') then etat := 5; end if;
                        when others => etat := 0;
                    end case;
                    if (etat = 5) then
                        LED_0 <= '1';
                    else
                        LED_0 <= '0';
                    end if;
                end if;
    end process;
end Behavioral;
```

FIGURE 13 – *Programme VHDL détecteur de code*

Afin de ne pas réinitialiser l'état de la machine avec une mauvaise entrée, on a enlevé les "else etat = 0" pour chaque cas en comparaison avec le code du premier détecteur.

4.4 Simulation du circuit et observation

Voici le chronogramme de la simulation que l'on a obtenu après avoir entré le code "110010" :

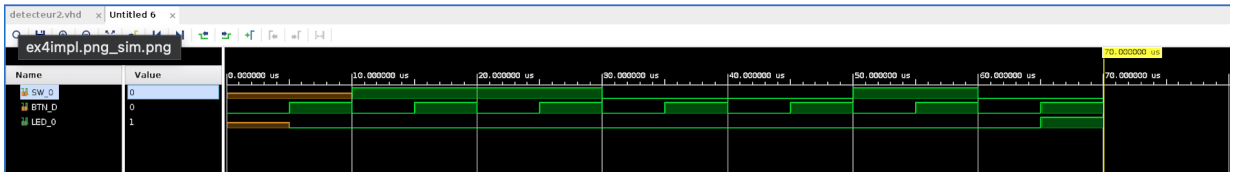


FIGURE 14 – *Schéma RTL détecteur de code avec fausses entrées négligées*

Nous avons volontairement entré un faux code ; "110010" au lieu de "11010", pour bien montrer que le circuit garde la valeur précédente et ne réinitialise pas le compteur en cas de mauvaise entrée. En effet, même après avoir entré le '0' en trop on peut remarquer que LED_0 est bien à 1.

4.5 Implémentation et schéma RTL

Après avoir lancé l'inplémentation, on obtient le schéma RTL suivant :

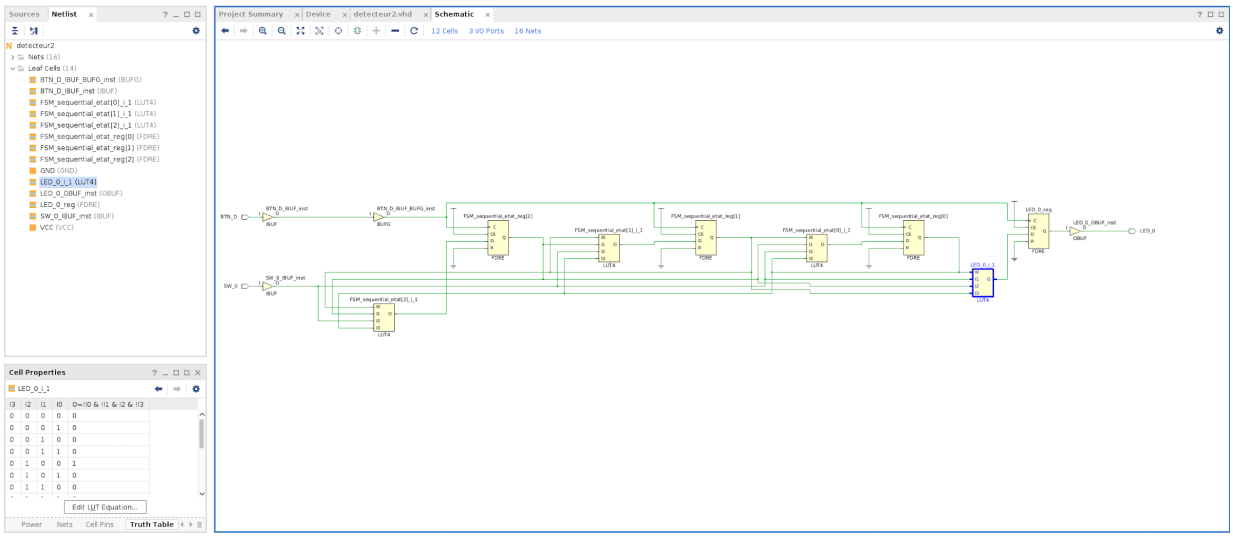


FIGURE 15 – *Schéma RTL détecteur de code*

Le schéma d'implémentation est le même que celui de l'exercice 3, il n'y a que les LUT qui changent puisque ce sont elles qui gèrent les changements d'état, elles doivent donc être modifiées pour que l'état ne repasse pas à 0 à chaque erreur mais reste le même.

4.6 Programmation du FPGA

Nous n'avons malheureusement pas eu le temps de prendre des photos pour le fonctionnement du détecteur sur la carte. Voici ce que l'on a observé :

On a tout d'abord essayé de directement mettre le code "11010" afin de voir si la LED 0 s'allumait. Ainsi, on a procédé de la manière suivante : on changeait d'abord SW_0 dans la position voulue (haut pour 1, bas pour 0), puis on appuyait brièvement sur BTN_D pour que le système change d'état. Ainsi, après avoir inséré le dernier 0, la LED 0 s'est allumée, ce qui montrait que la programmation avait bien fonctionné.

Ensuite, pour voir si la machine à état se réinitialise après une mauvaise entrée, on a réalisé le test suivant : on a inséré "110", puis "0" (erreur) , puis "10" (faire comme si l'erreur ne s'était pas passée). Contrairement au premier détecteur, la LED 0 s'est quand même allumée, ce qui montre bien que dans le cas de cette machine, une erreur ne réinitialise pas l'état.

4.7 Nombre de bits d'états et équation de sortie

Concernant le nombre de bits d'états, on ne peut pas juger étant donné que dans l'exercice 3 il n'y avait pas de bit d'états à proprement parlé. Dans ce cas ci on a 4 FDRE donc 4 bits d'état, logique car $2^4 > 5$ états.

Concernant l'équation de sortie, elle est beaucoup simple rien quà la lecture. En effet, dans l'exercice 3, le logiciel a traité l'implémentation avec la méthode One-hot, résultant des équations de sortie très compliquées. Ici on peut apercevoir sur le LUT4 l'équation : $\bar{I_0}$ & $\bar{I_1}$ & $\bar{I_2}$ & $\bar{I_3}$. Cette équation est plus simple et le résultat sera différent de celle de la première car les I_i ont changé en raison des LUT.

5 Conclusion

Nous avons donc pu voir grâce à ce TP plusieurs manières de mémoriser les états précédents, sous forme de machine à états ou non, et plusieurs manières de les synthétiser.

En effet, les deux premiers exercices nous ont permis d'une part de mieux comprendre le fonctionnement de la synthétisation à l'aide de FDCE et FDRE, qui sont très utiles lorsqu'on a besoin de se souvenir de la valeur de l'état précédent, et fonctionnent très bien avec des reset synchrones ou asynchrones.

De plus, les deux derniers exercices nous ont permis de voir d'une autre part les différentes méthodes utilisées pour synthétiser une machine à états. La méthode du compteur permet de gérer un nombre d'états très grand, mais nécessite de décoder le numéro de l'état pour générer les sorties extérieures. La méthode One-Hot permet de dériver la sortie directement du registre de l'état courant, mais elle nécessite d'avoir un registre par état donc elle n'est pas optimisée pour les cas où on aurait besoin de beaucoup d'états.