
MI01

*Structure d'un calculateur informatique*

# Rapport TP n°3

# VHDL Séquentiel, comptage du temps

Groupe : mi01a031

Pillis Julien / Galleze Rayane

*Semestre d'automne 2022*

# Introduction

Au cours de ce rapport nous allons présenter les résultats des exercices du TP 3 : VHDL Séquentiel et comptage du temps. Ce TP a pour objectif de compléter l'étude de l'utilité des machines à états du TP2. Cela se traduit par l'application de machines à états à un comptage du temps, avec ou sans prise en compte d'entrées extérieures telles que des capteurs ou des boutons (reset). Le TP a également pour objectif de nous introduire à la réduction de fréquence d'horloge du FPGA, parfois nécessaire, pour adapter le comportement des contrôleurs que nous souhaitons réaliser (contrainte de temps).

# Exercice 1 : Diviseur de fréquence

## Objectifs

Comprendre le fonctionnement du diviseur de fréquence, déterminer la valeur d'une fréquence de sortie en fonction de l'horloge d'entrée et appliquer le raisonnement étudié sur le FPGA.

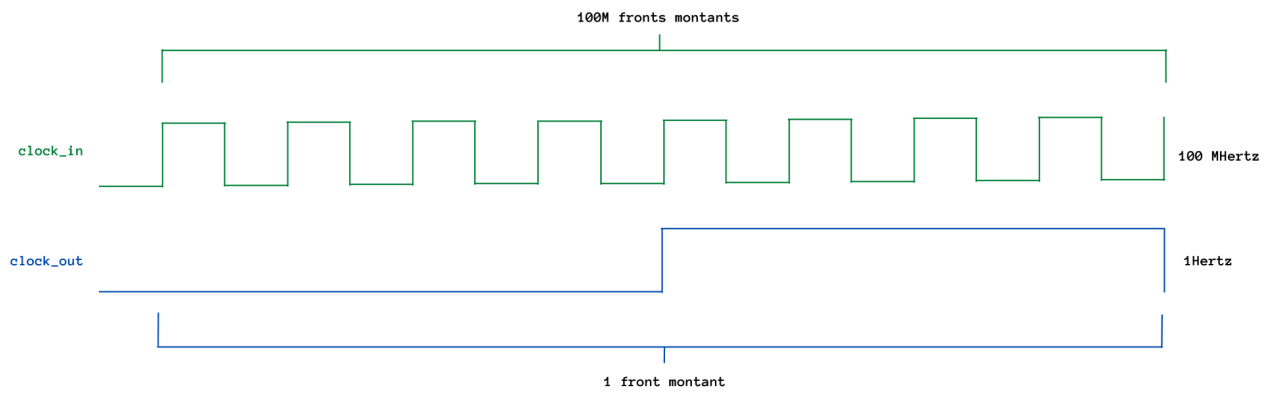
## Question 2

Le composant clock_divider reçoit en entrée un diviseur (entier dont la valeur est changeable). Bien évidemment, il reçoit également l'horloge dont la fréquence est à modifier (clock_in) et fournit la nouvelle fréquence clock_out.

Au sein de l'architecture, une variable c est instanciée (entier pouvant prendre les valeurs de 0 à divisor-1, et vaut 0 à l'initialisation). Il s'agit d'un compteur nous permettant de déterminer si, en sortie, notre composant envoie un front montant ("1") ou reste/envoie un front descendant à valeur "0" (sortie clock_out).

Pour diviser la fréquence d'entrée par le diviseur, il suffit simplement de diviser ce nombre diviseur-1 par 2 et d'incrémenter c de 1 à chaque front montant de clock_in, tant que c est inférieur à (diviseur-1)/2. La valeur de sortie de clock_out, lorsque c vérifie cette condition sera de "0". Si celle-ci n'est pas respectée, la valeur de clock_out sera alors toujours de "1", tant que c n'a pas subi *diviseur* incrémentations. Si ce nombre d'incrémentation est dépassé, on reset le compteur. Cela permet alors d'obtenir, à partir de n fronts montants de clock_in (100M dans le cadre du FPGA), un seul et plus long front montant sur clock_out pendant une seconde. Le diviseur de fréquence vérifie donc bien la relation : *f(clock_out) = f(clock_in) / divisor*

Voici un schéma explicatif (utilisation du générateur d'horloge du FPGA (f(clock_in) = 100MHz)) :




*NB : Un diviseur valant 100M a été utilisé pour obtenir une fréquence de sortie de 1Hz.*

## Question 3

Le générateur d'horloge du FPGA étant cadencé à 100 MHz (=f(clock_in)) et le diviseur valant 20 MHz, nous obtenons ainsi :

$$f(clock\_out) = f(clock\_in) / divisor = 100MHz / 20MHz = 5Hz$$

La fréquence du signal de sortie de cet exemple est donc de 5 Hz.

## Question 4

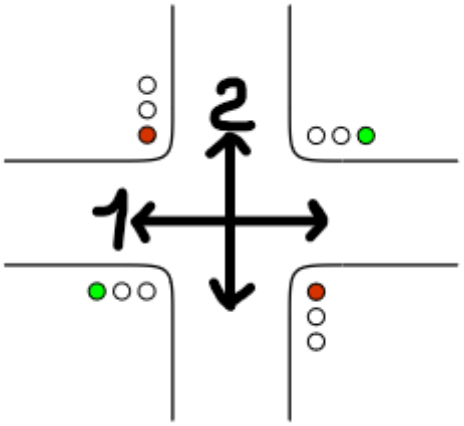
Après programmation du FPGA, le diviseur de fréquence était fonctionnel.

# Exercice 2 : Feux de circulation

## Objectifs

Mise en place d'un système de gestion de feux de circulation. Ce système sera développé pour être fonctionnel sur 2 axes et devra dans un premier temps ne pas prendre en compte de reset. Bien évidemment, le système sera synchrone et prendra en horloge d'entrée, celle du FPGA cadencée à 100 MHz.

Schéma de la situation




*1 : Axe de circulation 1 / 2 : Axe de circulation 2*

La gestion des feux sur un même axe est identique : Les feux de l'axe 1 seront rouges, verts et oranges pendant les mêmes périodes. De même pour les feux de l'axe 2. Cependant, il faudra veiller à respecter certaines règles. Par exemple : les feux des deux axes ne peuvent pas être rouges, verts ou oranges en même temps !

Pour cela, établissons ces règles dans la question 1.

## Question 1

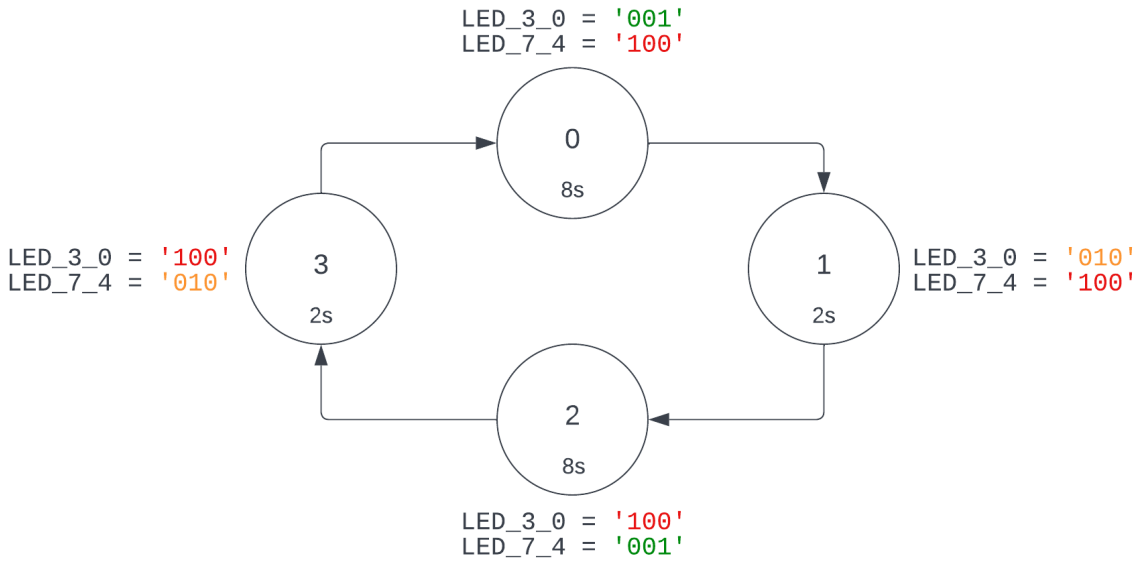
Si les feux d'un axe ne peuvent être rouges plus de 10 secondes, il faut que les feux de l'autre axe soient verts puis oranges en 10 secondes. Or, les feux ne peuvent être oranges plus de 2 secondes. Donc les feux seront verts pendant 8 secondes.

Ainsi, on en déduit la séquence suivante :

- Si les feux de l'axe 1 sont oranges pendant 2 secondes : les feux de l'axe 2 sont rouges pendant 2 secondes
- Si les feux de l'axe 1 sont rouges pendant 10 secondes : les feux de l'axe 2 sont verts pendant 8 secondes

- Si les feux de l'axe 2 sont oranges pendant 2 secondes : les feux de l'axe 1 sont rouges pendant 2 secondes
- Si les feux de l'axe 2 sont rouges 10 secondes pendant : les feux de l'axe 1 sont verts 8 secondes

Pour mieux comprendre cette séquence, la voici traduite sous forme de schéma :

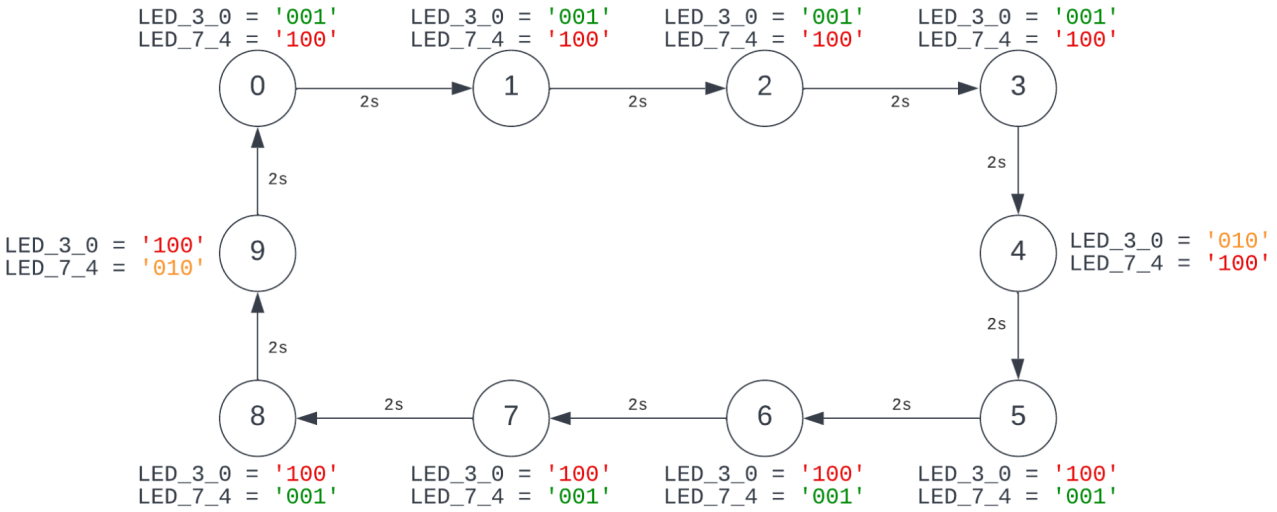



*NB : La machine à états correspondant à la séquence se trouve en Q2.*

## Question 2

D'après les durées des phases, on constate que la plus petite durée de phase est de 2 secondes ce qui représente la durée du feu orange avant de passer au vert. De plus, les autres durées sont des multiples de 2 (8 secondes pour le feu vert, 10 secondes pour le feu rouge). La période minimale devrait être ainsi égale au plus grand commun diviseur, soit 2 secondes. Ceci est équivalent à une fréquence de 0.5 Hz.

Pour obtenir cette fréquence, il suffira ainsi de diviser la fréquence initiale de l'horloge du FPGA par deux fois sa valeur, soit 200MHz.
Voici la machine à états correspondant à la séquence en prenant en compte la fréquence de l'horloge :




## Question 3

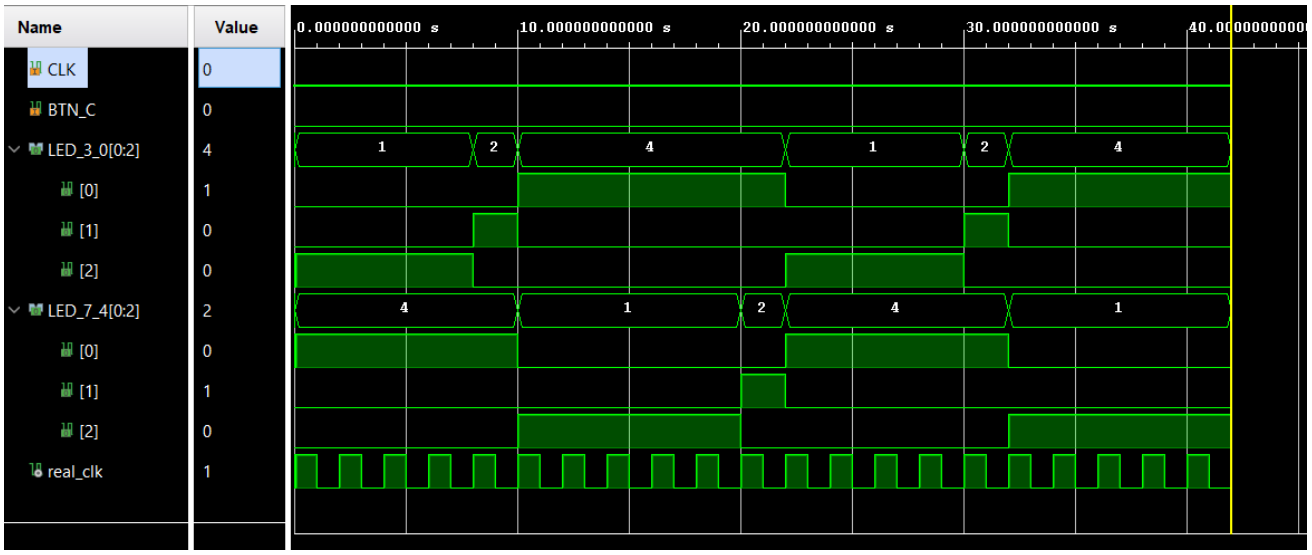
Modélisation VHDL du contrôleur : ***feu_tricolore_NR.vhd***

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity feu_tricolore_NR is
      port (CLK, BTN_C : in bit;
            LED_3_0, LED_7_4 : out integer range 1 to 4);
end feu_tricolore_NR;

architecture Behavioral of feu_tricolore_NR is
   component clock_divider
      generic(divisor : integer);
      port(clock_in, reset : in bit;
            clock_out : out bit);
      end component clock_divider;
      signal real_clk : bit;

begin
      CD :  clock_divider generic map(divisor => 200000000) port map(clock_in
=> CLK, reset => '0', clock_out =>real_clk);
process(real_clk)
      variable etat : integer range 0 to 10;
begin
      if(real_clk'event and real_clk = '1') THEN
      case etat is
            when 0 => LED_3_0 <= 1; LED_7_4 <= 4; etat:=1;
```

```
            when 1 => LED_3_0 <= 1; LED_7_4 <= 4; etat:=2;
            when 2 => LED_3_0 <= 1; LED_7_4 <= 4; etat:=3;
            when 3 => LED_3_0 <= 1; LED_7_4 <= 4; etat:=4;
            when 4 => LED_3_0 <= 2; LED_7_4 <= 4; etat:=5;
            when 5 => LED_3_0 <= 4; LED_7_4 <= 1; etat:=6;
            when 6 => LED_3_0 <= 4; LED_7_4 <= 1; etat:=7;
            when 7 => LED_3_0 <= 4; LED_7_4 <= 1; etat:=8;
            when 8 => LED_3_0 <= 4; LED_7_4 <= 1; etat:=9;
            when 9 => LED_3_0 <= 4; LED_7_4 <= 2;etat:=0;
            when others => etat:=4;
        end case;
        end if;
end process;
end Behavioral;
```

## Question 3

Voici le résultat de la simulation




La simulation du circuit montre que le fonctionnement souhaité du contrôleur est bien représenté. En effet, à chaque top d'horloge ⇔ période de 2 secondes, le système évolue. Au départ, nous sommes à l'état 0 puisque le feu de l'axe 1 (LED_3_0) est vert (vaut 1 donc 1ère LED allumée) et le feu de l'axe 2 (LED_7_4) est rouge (vaut 4 donc 3ème LED allumée) . Après 4 * 2s le feu de l'axe 1 (LED_3_0) passe au orange (vaut 2 donc 2ème LED allumée) et le feu de l'axe 2 (LED_7_4) reste au rouge ⇒ nous sommes bien allés jusqu'à l'état 4. 2 secondes plus tard, le feu de l'axe 1 passe au rouge et le feu de l'axe 2 passe au vert et ainsi de suite.

## Question 5

Après génération du BitStream et programmation du FPGA, le système était fonctionnel.

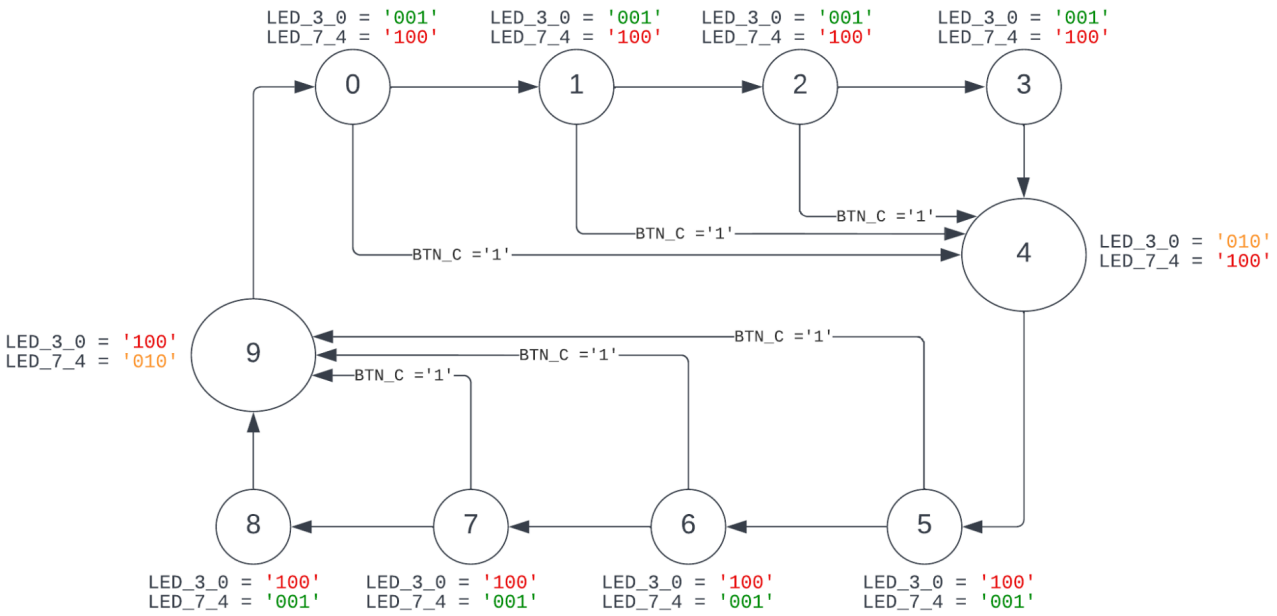
## Question 6

Intégrer une remise à zéro fiable du système signifie qu'il faut s'assurer qu'elle se fera en toute sécurité. Pour cela, nous devons assurer, lors d'une demande de reset, qu'un feu vert ne devienne pas rouge immédiatement, mais passe par l'orange, et qu'un feu rouge ne devienne pas vert de suite. Pour ne pas perturber les périodes (mais aussi les automobilistes) nous avons estimé qu'il était préférable d'implémenter un **reset synchrone** (dépendant donc de l'horloge).

Ainsi, lorsque la machine à états se trouve dans les états 0, 1, 2 ou 3, il est nécessaire de passer à l'état 4 au prochain top d'horloge si le reset est demandé.
De la même façon, lorsque la machine à états se trouve dans les états 5, 6, 7 ou 8, il est nécessaire de passer à l'état 9 au prochain top d'horloge si le reset est demandé.

Ici, il n'est pas nécessaire de faire une demande de reset lorsque la machine se trouve dans les états 3 ou 8, puisqu'elle passera directement aux états de remise à 0 lors de la prochaine période.

Voici la machine à états avec intégration d'un reset synchrone de remise à zéro fiable :



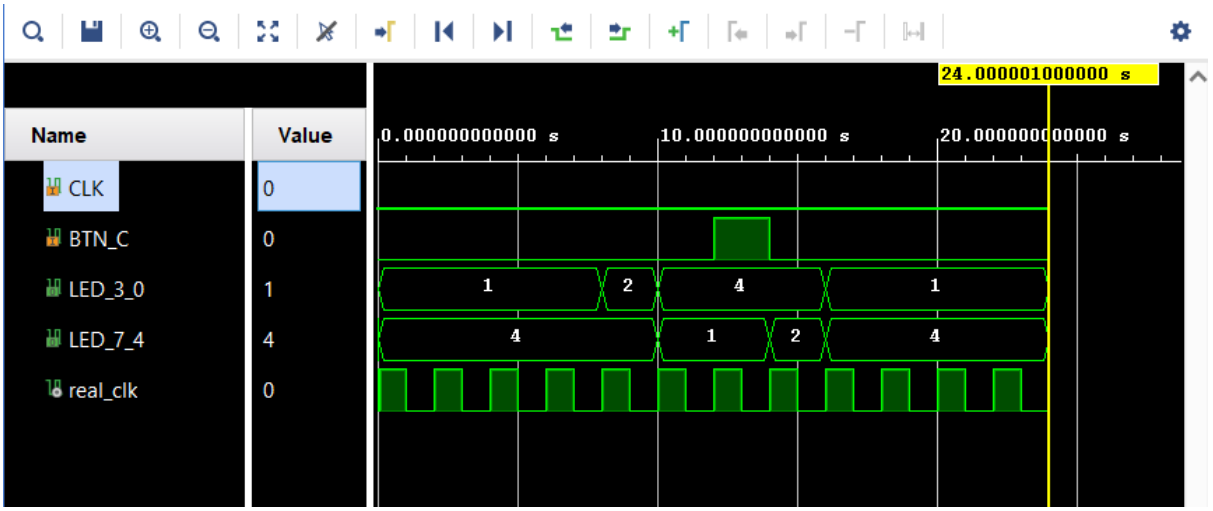
## Question 7

Modélisation VHDL du système avec reset synchrone : ***feu_tricolore_R.vhd***

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity feu_tricolore_R is
      port (CLK, BTN_C : in bit;
            LED_3_0, LED_7_4 : out integer range 1 to 4);
end feu_tricolore_R;

architecture Behavioral of feu_tricolore_R is
   component clock_divider
      generic(divisor : integer);
      port(clock_in, reset : in bit;
            clock_out : out bit);
      end component clock_divider;
      signal real_clk : bit;

begin
      CD :  clock_divider generic map(divisor => 200000000) port
map(clock_in => CLK, reset => '0', clock_out =>real_clk);
process(real_clk)
      variable etat : integer range 0 to 10;
begin
      if(real_clk'event and real_clk = '1') THEN
      case etat is
            when 0 => LED_3_0 <= 1; LED_7_4 <= 4;
                  if(BTN_C = '1') then etat := 4;
                  else  etat:=1;
                  end if;
            when 1 => LED_3_0 <= 1; LED_7_4 <= 4;
                  if(BTN_C = '1') then etat := 4;
                  else  etat:=2;
                  end if;
            when 2 => LED_3_0 <= 1; LED_7_4 <= 4;
                  if(BTN_C = '1') then etat := 4;
                  else  etat:=3;
                  end if;
            when 3 => LED_3_0 <= 1; LED_7_4 <= 4; etat:=4;
            when 4 => LED_3_0 <= 2; LED_7_4 <= 4;
                  if(BTN_C = '0') then etat := 5;
                  end if;
            when 5 => LED_3_0 <= 4; LED_7_4 <= 1;
                  if(BTN_C = '1') then etat := 9;
```

```
                else  etat:=6;
                end if;
            when 6 => LED_3_0 <= 4; LED_7_4 <= 1;
                if(BTN_C = '1') then etat := 9;
                else  etat:=7;
                end if;
            when 7 => LED_3_0 <= 4; LED_7_4 <= 1;
                if(BTN_C = '1') then etat := 9;
                else  etat:=8;
                end if;
            when 8 => LED_3_0 <= 4; LED_7_4 <= 1; etat:=9;
            when 9 => LED_3_0 <= 4; LED_7_4 <= 2;
                if(BTN_C = '0') then etat := 0;
                end if;
            when others => etat:=4;
        end case;
        end if;
end process;
end Behavioral;
```

Voici le résultat de la simulation




La simulation du circuit montre que le fonctionnement souhaité du contrôleur est bien représenté. En effet, à chaque top d'horloge ⇔ période de 2 secondes, le système évolue de la même manière que le système sans reset. Cependant, lorsque que BTN_C (le reset) est appuyé/activé, la machine à états retournera dans l'état correspondant à la remise à 0 en fonction de son état actuel. Dans le cas de la simulation, nous avons effectué l'appui sur le reset lorsque la machine se trouve à l'état 6. A la prochaine période, elle se trouve à l'état 9 puis reprend à l'état 0. Le reset a donc été correctement effectué.

Après génération du BitStream et programmation du FPGA, le système était fonctionnel

# Exercice 3 : Prise en compte d'un capteur de voiture

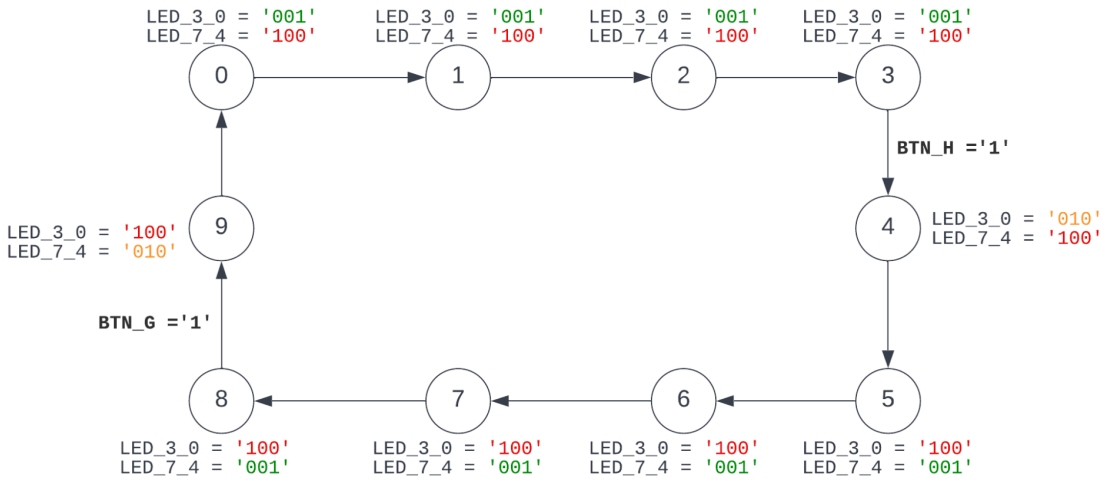
## Question 1

Initialement, le système se comporte comme celui de l'exercice 2. Cependant, nous devons intégrer la détection de capteurs. Ces capteurs, présents au niveau des feux de circulation, viendront influencer la durée des feux verts et rouges. Tant que le détecteur de l'axe où les feux sont rouges ne détecte pas de présence, les feux de l'autre axe restent verts. Si le capteur s'active, les feux de l'axe où les feux sont verts devront passer au orange, si les 8 secondes minimum ont déjà été passées. Dans le cas échéant, ils devront finir les 8 secondes, puis passer au orange.

Ainsi, on en déduit la séquence suivante :

- Si les feux de l'axe 1 sont oranges pendant 2 secondes : les feux de l'axe 2 sont rouges pendant 2 secondes
- Si les feux de l'axe 1 sont rouges pendant 10 secondes : les feux de l'axe 2 sont verts pendant 8 secondes
    - Si le capteur de l'axe 1, détecte une présence, alors les feux verts passeront directement au orange à la période suivant celle des 8 secondes
    - Sinon les feux restent verts jusqu'à détection de présence
- Si les feux de l'axe 2 sont oranges pendant 2 secondes : les feux de l'axe 1 sont rouges pendant 2 secondes
- Si les feux de l'axe 2 sont rouges 10 secondes pendant : les feux de l'axe 1 sont verts 8 secondes
    - Si le capteur de l'axe 2, détecte une présence, alors les feux verts passeront directement au orange à la période suivant celle des 8 secondes
    - Sinon les feux restent verts jusqu'à détection de présence

Ce fonctionnement se traduit par la machine à états suivante

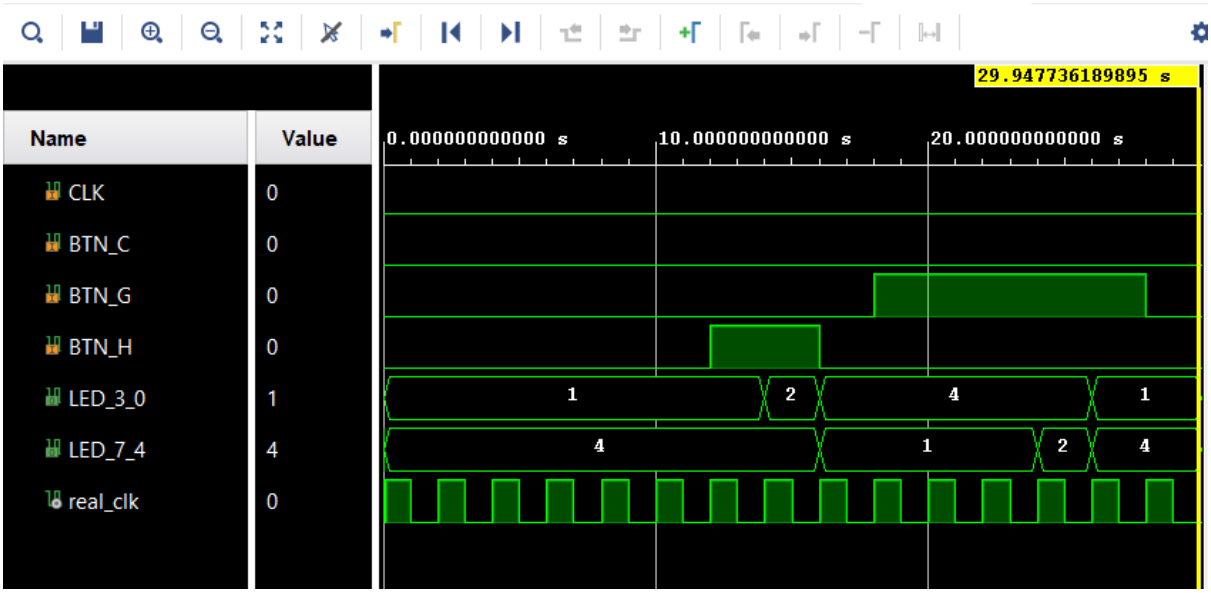


## Question 2

Voici la modélisation VHDL du système : ***capteur_NR.vhd***

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity capteur_NR is
      port (CLK, BTN_C, BTN_G, BTN_H: in bit;
            LED_3_0, LED_7_4 : out integer range 1 to 4);
end capteur_NR;

architecture Behavioral of capteur_NR is
   component clock_divider
      generic(divisor : integer);
      port(clock_in, reset : in bit;
            clock_out : out bit);
      end component clock_divider;
      signal real_clk : bit;

begin
      CD :  clock_divider generic map(divisor => 200000000) port
map(clock_in => CLK, reset => BTN_C, clock_out =>real_clk);
process(real_clk)
      variable etat : integer range 0 to 10;
begin
      if(real_clk'event and real_clk = '1') THEN
      case etat is
            when 0 => LED_3_0 <= 1; LED_7_4 <= 4; etat:=1;
            when 1 => LED_3_0 <= 1; LED_7_4 <= 4; etat:=2;
            when 2 => LED_3_0 <= 1; LED_7_4 <= 4; etat:=3;
            when 3 => LED_3_0 <= 1; LED_7_4 <= 4;
                  if BTN_H = '1' then etat:=4;
                  end if;
            when 4 => LED_3_0 <= 2; LED_7_4 <= 4; etat:=5;
            when 5 => LED_3_0 <= 4; LED_7_4 <= 1; etat:=6;
            when 6 => LED_3_0 <= 4; LED_7_4 <= 1; etat:=7;
            when 7 => LED_3_0 <= 4; LED_7_4 <= 1; etat:=8;
            when 8 => LED_3_0 <= 4; LED_7_4 <= 1;
                  if BTN_G = '1' then etat := 9;
                  end if;
            when 9 => LED_3_0 <= 4; LED_7_4 <= 2;etat:=0;
            when others => etat:=4;
      end case;
      end if;
end process;
end Behavioral;
```

## Question 3




La simulation du circuit montre que le fonctionnement souhaité du système est bien représenté. En effet, celui-ci fonctionne de manière identique à celui de l'exercice 2 si aucun des capteurs (BTN_G et BTN_H) ne détecte de présence (valent 1). On observe ainsi qu'en l'absence de présence, les feux verts restent verts (LED_3_0), et les feux rouges restent rouges (LED_7_4). Cependant, si le capteur de l'axe 2 envoie un signal et que les feux de l'axe 1 sont verts (LED_3_0 à 1), alors le feu passe au orange à la prochaine période puis laissera les feux de l'axe 2 (LED_7_4) devenir verts. En effet, la période minimale des 8 secondes ici est largement dépassée (12 secondes au moment de la détection).
On remarque également que lorsque le capteur BTN_G s'active alors que la période minimale des feux verts de l'axe 2 n'est pas dépassée, celle-ci termine ses 8 secondes avant que les feux ne passent au orange.

Le système vérifie la séquence voulue et est donc fonctionnel.

## Question 4

Après génération du BitStream et programmation du FPGA, le système était fonctionnel.

## Question 5

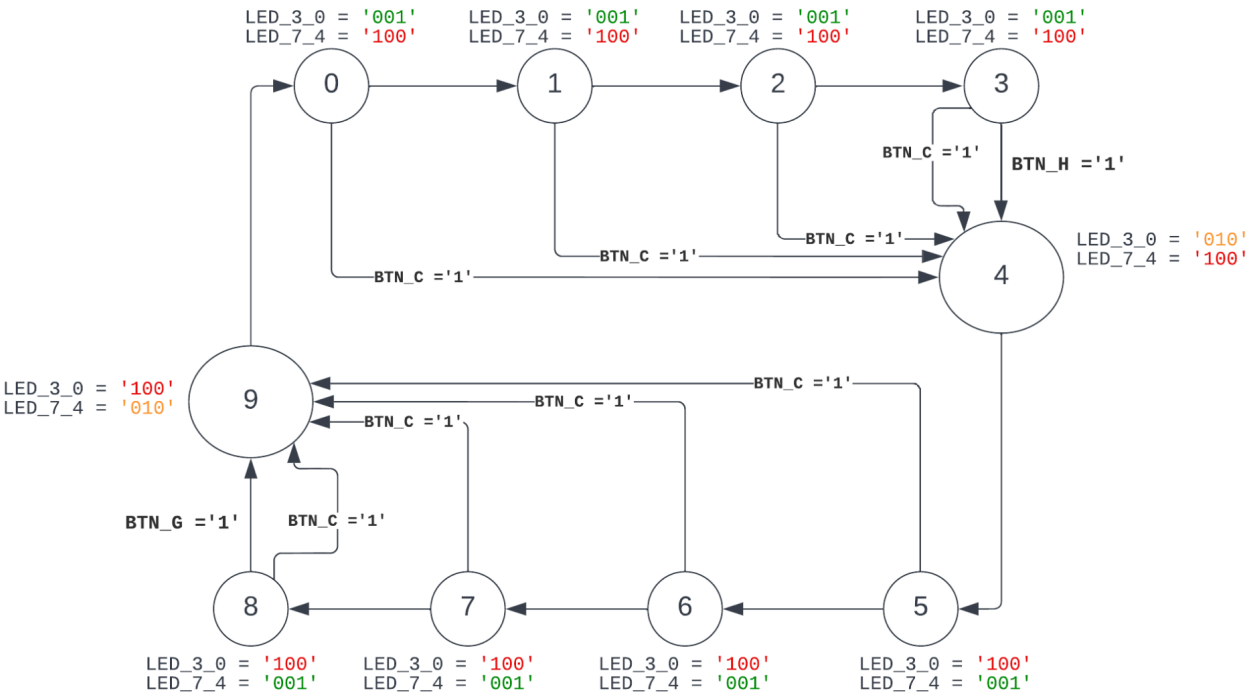
De la même manière que lors de l'exercice 2, nous devons assurer que, lors d'une demande de reset, un feu vert ne devienne pas rouge immédiatement, mais passe par l'orange, et qu'un feu rouge ne devienne pas vert de suite. Pour ne pas perturber les périodes (mais

aussi les automobilistes) nous avons estimé qu'il était préférable d'implémenter un **reset synchrone** (dépendant donc de l'horloge).

Ainsi, lorsque la machine à états se trouve dans les états 0, 1, 2 ou 3, il est nécessaire de passer à l'état 4 au prochain top d'horloge si le reset est demandé.
De la même façon, lorsque la machine à états se trouve dans les états 5, 6, 7 ou 8, il est nécessaire de passer à l'état 9 au prochain top d'horloge si le reset est demandé.

Cependant, **il faudra ici laisser la possibilité de faire un reset lorsque la machine à états se trouve dans l'état 3 ou 8** puisqu'elle ne passe pas, respectivement, à l'état 4 et 9 au prochain top d'horloge, mais uniquement si une présence est détectée sur le capteur correspondant.

Voici donc la machine à états correspondante



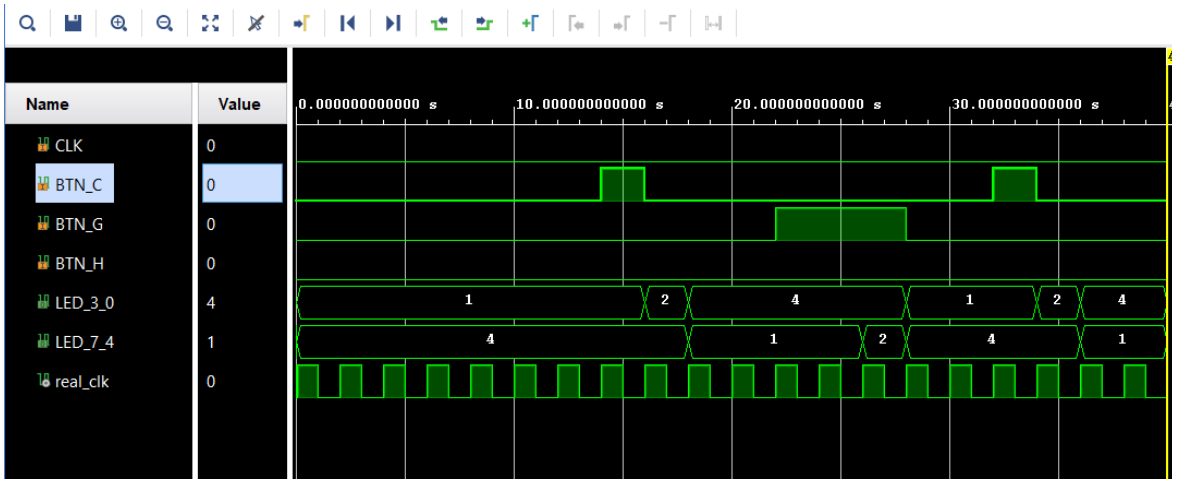
## Question 6

Modélisation VHDL du système avec reset synchrone : ***capteur_R.vhd***

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity capteur_R is
      port (CLK, BTN_C, BTN_G, BTN_H: in bit;
            LED_3_0, LED_7_4 : out integer range 1 to 4);
end capteur_R;

architecture Behavioral of capteur_R is
   component clock_divider
      generic(divisor : integer);
      port(clock_in, reset : in bit;
            clock_out : out bit);
      end component clock_divider;
      signal real_clk : bit;

begin
      CD :  clock_divider generic map(divisor => 200000000) port
map(clock_in => CLK, reset => '0', clock_out =>real_clk);
process(real_clk)
      variable etat : integer range 0 to 10;
begin
      if(real_clk'event and real_clk = '1') THEN
      case etat is
            when 0 => LED_3_0 <= 1; LED_7_4 <= 4;
                 if(BTN_C = '1') then etat := 4;
                else  etat:=1;
                end if;
            when 1 => LED_3_0 <= 1; LED_7_4 <= 4;
                 if(BTN_C = '1') then etat := 4;
                else  etat:=2;
                end if;
            when 2 => LED_3_0 <= 1; LED_7_4 <= 4;
                 if(BTN_C = '1') then etat := 4;
                else  etat:=3;
                end if;
            when 3 => LED_3_0 <= 1; LED_7_4 <= 4;
                  if BTN_H = '1' OR BTN_C = '1' then etat:=4;
                  end if;
            when 4 => LED_3_0 <= 2; LED_7_4 <= 4;
                 if(BTN_C = '0') then etat := 5;
                end if;
            when 5 => LED_3_0 <= 4; LED_7_4 <= 1;
```

```
                if(BTN_C = '1') then etat := 9;
               else  etat:=6;
               end if;
          when 6 => LED_3_0 <= 4; LED_7_4 <= 1;
                if(BTN_C = '1') then etat := 9;
               else  etat:=7;
               end if;
          when 7 => LED_3_0 <= 4; LED_7_4 <= 1;
                if(BTN_C = '1') then etat := 9;
               else  etat:=8;
          end if;
          when 8 => LED_3_0 <= 4; LED_7_4 <= 1;
                 if BTN_G = '1' OR BTN_C = '1' then etat := 9;
                 end if;
          when 9 => LED_3_0 <= 4; LED_7_4 <= 2;etat:=0;
          when others => etat:=4;
      end case;
      end if;
end process;
end Behavioral;
```

Simulation du système




La simulation du circuit montre que le fonctionnement souhaité du système est bien représenté. En effet, celui-ci fonctionne de manière identique à celui sans reset. Si on effectue un reset lorsque la machine à états se trouve dans l'état 3 et après avoir dépassé la période minimale des feux verts (14 secondes pendant la simulation) , les feux passent au orange à la prochaine période, de la même manière que si la machine se trouve dans les autres états.

Après génération du BitStream et programmation du FPGA, le système était fonctionnel

# Conclusion

Pour conclure ce rapport, nous pouvons ainsi affirmer que nous avons pu développer notre prise en main de la modélisation séquentielle en VHDL à l'aide du logiciel Vivado lors de ce TP. Nous avons notamment pu comprendre comment diviser la fréquence d'une horloge initiale, pour obtenir une fréquence recherchée et adaptée au cas pratique du TP. Les exercices nous ont permis d'apprendre beaucoup d'informations sur la façon d'implémenter la notion de cadence d'horloge sur nos machines à états et modélisations, et sur la façon de coupler un reset de système avec un système séquentiel nécessitant une remise à zéro fiable.