

# UNIVERSITÉ DE TECHNOLOGIE DE COMPIÈGNE

## MI01 - STRUCTURE D'UN CALCULATEUR

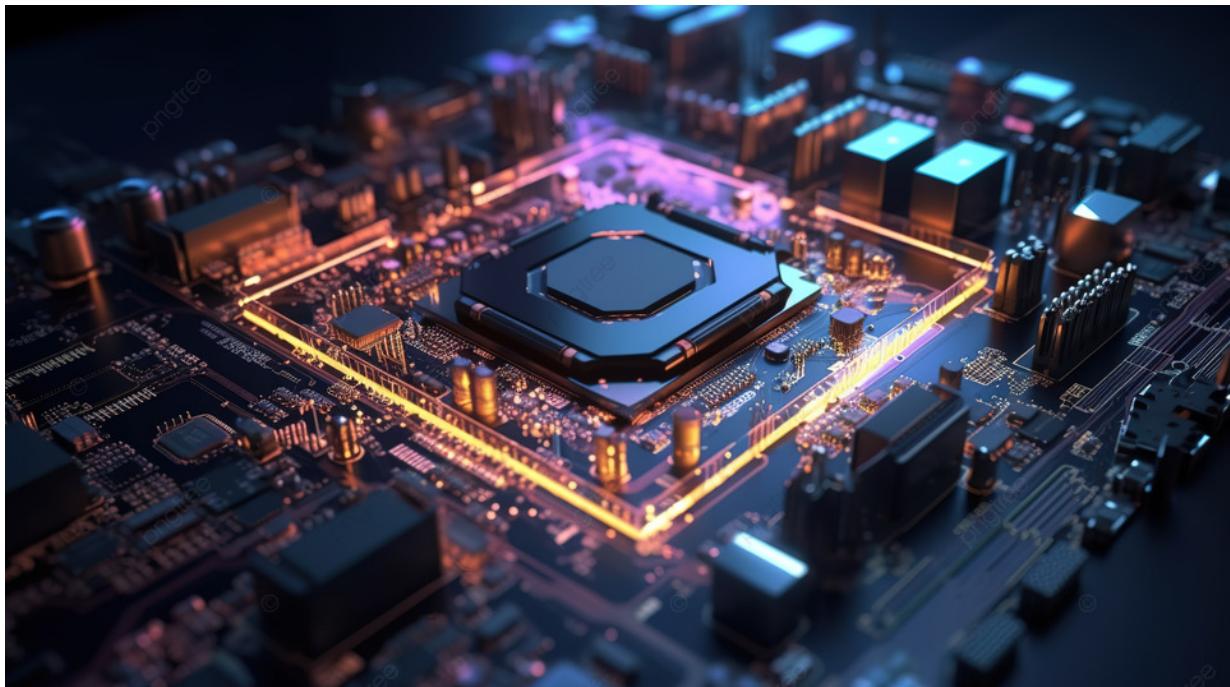
Responsable  
Mr. Shawky Marc

---

## TP3 - VHDL SEQUENTIEL, COMPTAGE DU TEMPS

---

Responsable TP  
Mr. Sanahuja Guillaume



# Sommaire

---

<b>1 Exercice 1 - Diviseur de fréquence</b>	<b>3</b>
1.1 OBJECTIF . . . . .	3
1.2 EXPLICATION SUCCINCTE DU FONCTIONNEMENT DU DIVISEUR DE FRÉQUENCE . . . . .	3
1.3 FRÉQUENCE DU SIGNAL DE SORTIE . . . . .	3
1.4 PROGRAMMATION DU FPGA . . . . .	3
<b>2 Exercice 2 - Feux de Circulation</b>	<b>4</b>
2.1 OBJECTIF . . . . .	4
2.2 DURÉE DES PHASES ET DÉROULEMENT DES SÉQUENCES	4
2.3 FRÉQUENCE MINIMALE DE L'HORLOGE . . . . .	5
2.4 MODÉLISATION DU CONTRÔLEUR EN VHDL . . . . .	5
2.5 SIMULATION DES FEUX DE CIRCULATION . . . . .	7
2.6 PROGRAMMATION DU FPGA . . . . .	8
2.7 REMISE À ZÉRO FIABLE DU SYSTÈME . . . . .	8
2.8 MODÉLISATION DU FEUX AVEC UNE REMISE À ZÉRO FIABLE EN VHDL . . . . .	9
2.9 SIMULATION AVEC RESET SYNCHRONE . . . . .	11
<b>3 Exercice 3 - Prise en compte d'un capteur de voiture</b>	<b>11</b>
3.1 DÉROULEMENT DE LA NOUVELLE SÉQUENCE . . . . .	11
3.2 MACHINE À ÉTAT DU FEUX AVEC CAPTEUR SANS RESET	13
3.3 MODÉLISATION DU SYSTÈME EN VHDL . . . . .	13
3.4 SIMULATION AVEC CAPTEURS . . . . .	15
3.5 PROGRAMMATION DU FPGA . . . . .	15
3.6 FEUX AVEC CAPTEUR ET RESET SYNCHRONE . . . . .	15
3.7 MACHINE À ÉTATS DU SYSTÈME . . . . .	16
3.8 MODÉLISATION DU SYSTÈME EN VHDL . . . . .	16
3.9 SIMULATION AVEC CAPTEURS ET RESET SYNCHRONE . .	18
<b>4 Conclusion</b>	<b>19</b>

## Table des figures

---

1	<i>Axes de circulation</i>	4
2	<i>Machine à états feux de circulation</i>	5
3	<i>Simulation feux de circulation</i>	7
4	<i>Machine à états du feux avec intégration d'un reset synchrone</i>	8
5	<i>Simulation feux avec reset</i>	11
6	<i>Machine à états feux avec capteurs</i>	13
7	<i>Simulation feux avec capteurs</i>	15
8	<i>Machine à états feux avec capteurs et reset synchrone</i>	16
9	<i>Simulation feux avec capteurs et reset</i>	18

# 1 Exercice 1 - Diviseur de fréquence

## 1.1 Objectif

L'objectif de cet exercice est de comprendre le fonctionnement du diviseur de fréquence, de déterminer la valeur d'une fréquence de sortie en fonction de l'horloge d'entrée et d'appliquer le raisonnement étudié sur le FPGA.

## 1.2 Explication succincte du fonctionnement du diviseur de fréquence

Le composant `clock_divider` reçoit en entrée un diviseur (entier dont la valeur est modifiable). 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  $(\text{diviseur}-1)/2$ . 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'incrémantation 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 :

$$f(\text{clock\_out}) = \frac{f(\text{clock\_in})}{\text{divisor}}$$

## 1.3 Fréquence du signal de sortie

Le générateur d'horloge du FPGA étant réglé sur une fréquence de 100 MHz ( $f(\text{clock\_in})$ ) et le diviseur sur 20 MHz, nous obtenons ainsi :

$$f(\text{clock\_out}) = \frac{f(\text{clock\_in})}{\text{divisor}} = \frac{100\text{MHz}}{20\text{MHz}} = 5\text{Hz}$$

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

## 1.4 Programmation du FPGA

Après programmation du FPGA, le diviseur de fréquence était fonctionnel. En effet, nous pouvions voir que la sortie `LED_0`, associé à `clock_out`, clignotait 5 fois en 1 seconde. Ce qui correspond à une période de 0.2s entre chaque clignotement. On retrouve bien à partir de ce résultat la fréquence calculée précédemment :

$$f(\text{clock\_out}) = \frac{1}{T_{\text{clock\_out}}} = \frac{1}{0.2} = 5\text{Hz}$$

## 2 Exercice 2 - Feux de Circulation

### 2.1 Objectif

Dans cet exercice, nous allons réaliser un contrôleur synchrone pour les feux d'un carrefour à deux axes principaux . Les deux axes sont symétriques, la durée de la phase rouge est de 10 secondes, la durée de la phase orange est de 2 secondes.

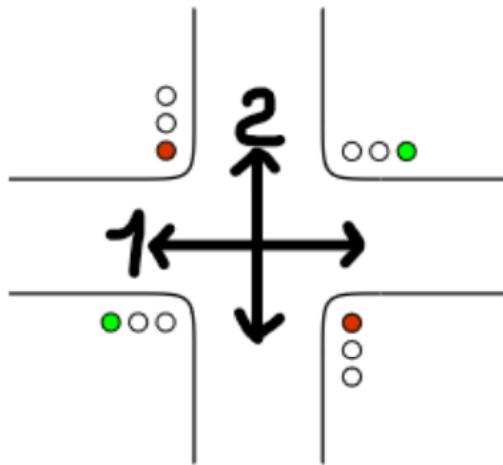


FIGURE 1 – Axes de circulation

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. Dans toute la suite du rapport nous considérerons sur les schémas des machines à états F1 qui correspond à LED\_3\_0 et F2 qui correspond à LED\_7\_4.

### 2.2 Durée des phases et déroulement des séquences

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 pendant 10 secondes ; les feux de l'axe 1 sont verts pendant 8 secondes

## 2.3 Fréquence minimale de l'horloge

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 :

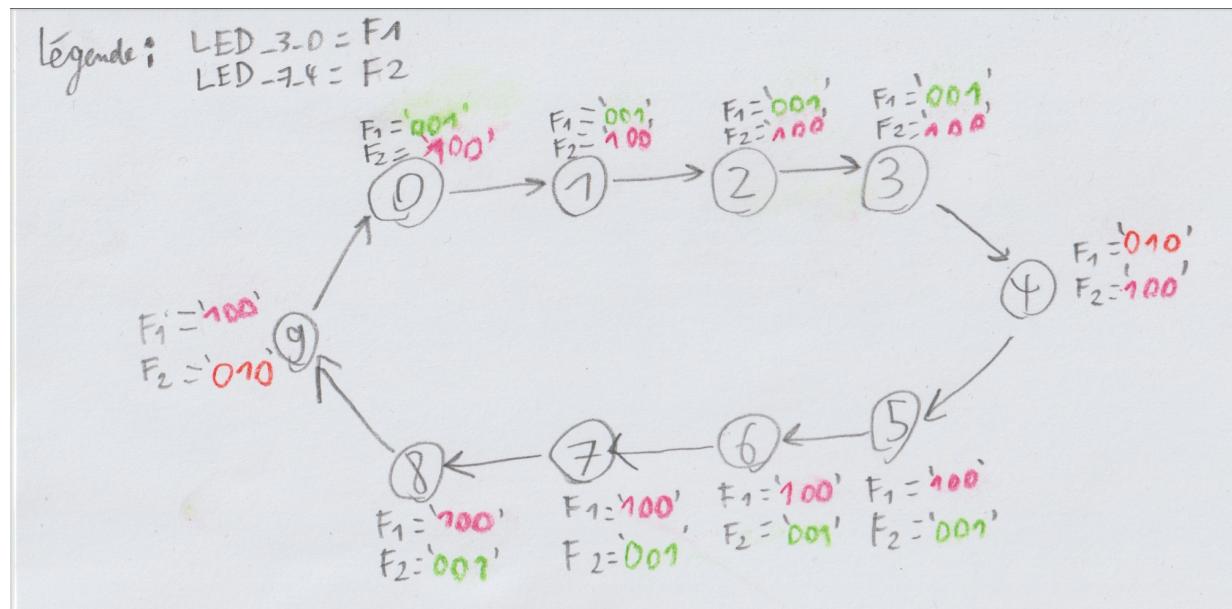


FIGURE 2 – Machine à états feux de circulation

## 2.4 Modélisation du contrôleur en VHDL

Nous avons écrit le code suivant dans le fichier feux.vhd :

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity feux is
    Port(CLK,BTN_C : in bit;
          LED_3_0,LED_7_4 : out integer range 0 to 15);
end feux;

architecture Behavioral of feux is
    component clock_divider
        generic(divisor : integer);
        port(clock_in, reset : in bit;
              clock_out : out bit);
    end component clock_divider;
    Signal CLK2 : bit;

```

```

begin
    CD : clock_divider
        generic map(divisor => 200000000)
        port map(clock_in => CLK, reset => BTN_C, clock_out => CLK2);

process(CLK2)
    variable etat : integer range 0 to 10 := 0;
begin
    if(CLK2'event and CLK2='1') then
        case etat is
            when 0 => etat:=1;
            when 1 => etat:=2;
            when 2 => etat:=3;
            when 3 => etat:=4;
            when 4 => etat:=5;
            when 5 => etat:=6;
            when 6 => etat:=7;
            when 7 => etat:=8;
            when 8 => etat:=9;
            when 9 => etat:=0;
            when others => etat:=0;
        end case;
        if (etat=0) then
            LED_3_0 <= 1; LED_7_4 <= 4;
        end if;
        if (etat=1) then
            LED_3_0 <= 1; LED_7_4 <= 4;
        end if;
        if (etat=2) then
            LED_3_0 <= 1; LED_7_4 <= 4;
        end if;
        if (etat=3) then
            LED_3_0 <= 1; LED_7_4 <= 4;
        end if;
        if (etat=4) then
            LED_3_0 <= 2; LED_7_4 <= 4;
        end if;
        if (etat=5) then
            LED_3_0 <= 4; LED_7_4 <= 1;
        end if;
        if (etat=6) then
            LED_3_0 <= 4; LED_7_4 <= 1;
        end if;
        if (etat=7) then
            LED_3_0 <= 4; LED_7_4 <= 1;
        end if;
        if (etat=8) then

```

```

        LED_3_0 <= 4; LED_7_4 <= 1;
    end if;
    if (etat=9) then
        LED_3_0 <= 4; LED_7_4 <= 2;
    end if;
end if;
end process;
end Behavioral;

```

Nous sommes contraints par le logiciel d'utiliser des groupes de 4 LEDs pour modéliser les 3 phases de chaque feux. Nous décidons volontairement de ne pas utiliser à chaque fois la LED qui a la position la plus élevée. Donc pour LED\_7\_4 nous n'utiliserons pas LED\_7\_4[3] (soit LED\_7) et pour LED\_3\_0 nous n'utiliserons pas LED\_3\_0[3] (soit LED\_3).

## 2.5 Simulation des feux de circulation

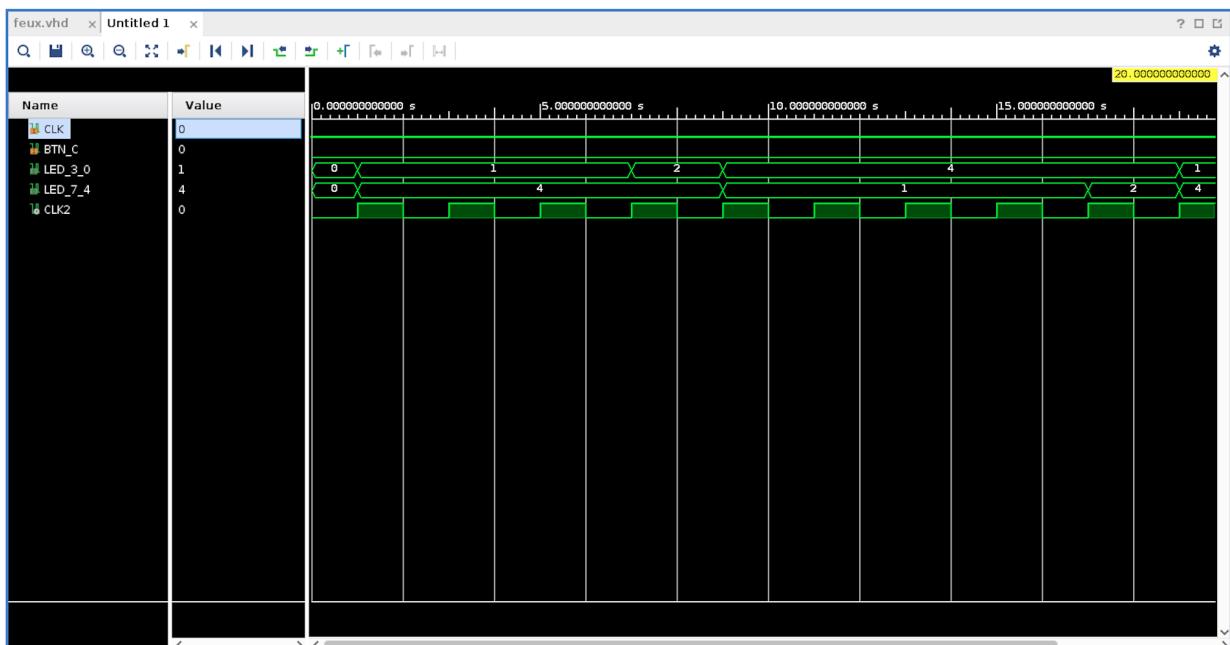


FIGURE 3 – *Simulation feux de circulation*

La simulation du circuit montre que le fonctionnement souhaité du contrôleur est bien représenté. En effet, les 2 signaux de sortie sont à 0 puis 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 LED\_3\_0[0] est allumée) et le feu de l'axe 2 (LED\_7\_4) est rouge (vaut 4 donc LED\_7\_4[2] est allumée) . Après 8s le feu de l'axe 1 (LED\_3\_0) passe au orange (vaut 2 donc LED\_3\_0[1] est 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 (vaut 4 donc LED\_3\_0[2] est allumée) et le feu de l'axe 2 passe au vert (vaut 1 donc LED\_7\_4[0] est allumée) et ainsi de suite.

## 2.6 Programmation du FPGA

Après génération du BitStream et programmation du FPGA, le système était fonctionnel. En effet, nous pouvions observer les différentes étapes et les bonnes périodes à travers les changements d'état qui se traduisait par un changement périodique des LEDs.

## 2.7 Remise à zéro fiable du système

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 :

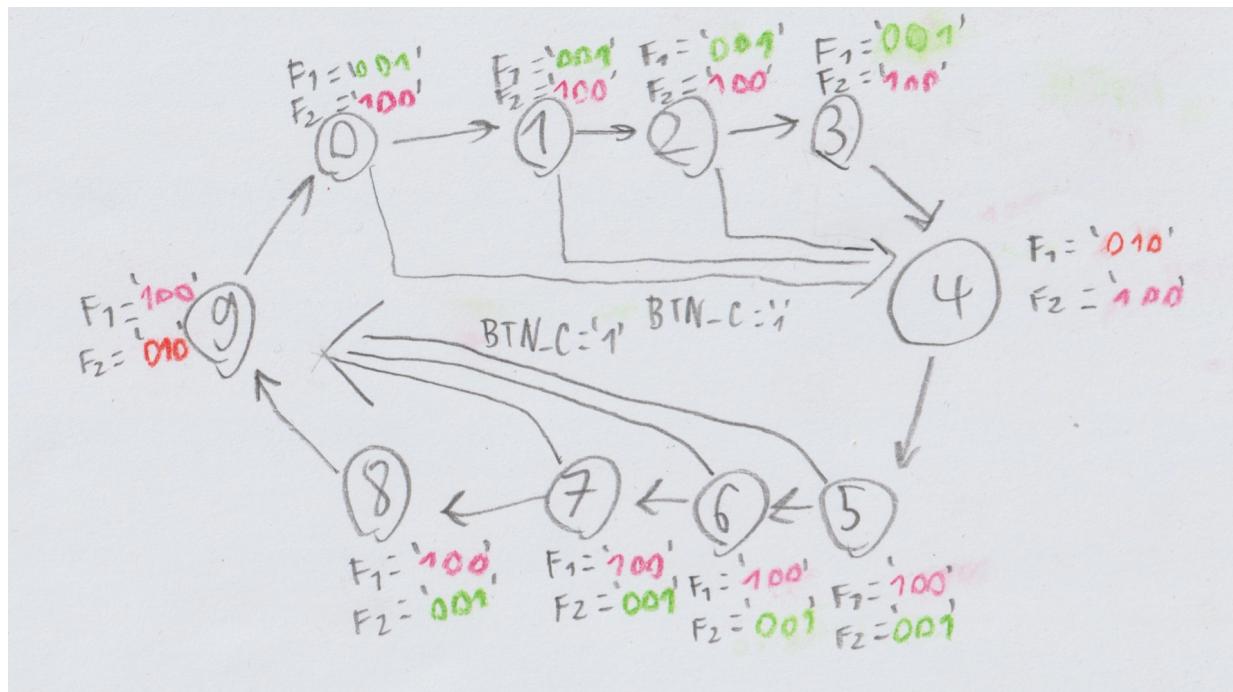


FIGURE 4 – Machine à états du feux avec intégration d'un reset synchrone

## 2.8 Modélisation du feux avec une remise à zéro fiable en VHDL

Nous avons écrit le code suivant dans le fichier feux\_R.vhd :

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

entity feux_R is
    Port(CLK,BTN_C : in bit;
          LED_3_0,LED_7_4 : out integer range 0 to 15);
end feux_R;

architecture Behavioral of feux_R is
    component clock_divider
        generic(divisor : integer);
        port(clock_in, reset : in bit;
              clock_out : out bit);
    end component clock_divider;
    Signal CLK2 : bit;

begin
    CD : clock_divider
        generic map(divisor => 200000000)
        port map(clock_in => CLK, reset => BTN_C, clock_out => CLK2);

    process(CLK2)
        variable etat : integer range 0 to 10 := 0;
    begin
        if(BTN_C='1') then
            if (etat=0 or etat=1 or etat=2 or etat=3 or etat=4) then
                etat :=0;LED_3_0 <= 1; LED_7_4 <= 4;
            else
                if(etat=5 or etat=6 or etat=7 or etat=8 or etat=9) then
                    etat := 9;LED_3_0 <= 4; LED_7_4 <= 2;
                end if;
            end if;
        else
            if(CLK2'event and CLK2='1') then
                case etat is
                    when 0 => etat:=1;
                    when 1 => etat:=2;
                    when 2 => etat:=3;
                    when 3 => etat:=4;
                    when 4 => etat:=5;
                    when 5 => etat:=6;
                    when 6 => etat:=7;
                    when 7 => etat:=8;
                    when 8 => etat:=9;
                    when 9 => etat:=0;
                end case;
            end if;
        end if;
    end process;
end;
```

```

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

## 2.9 Simulation avec reset synchrone

Voici le chronogramme de la simulation que l'on a obtenu :

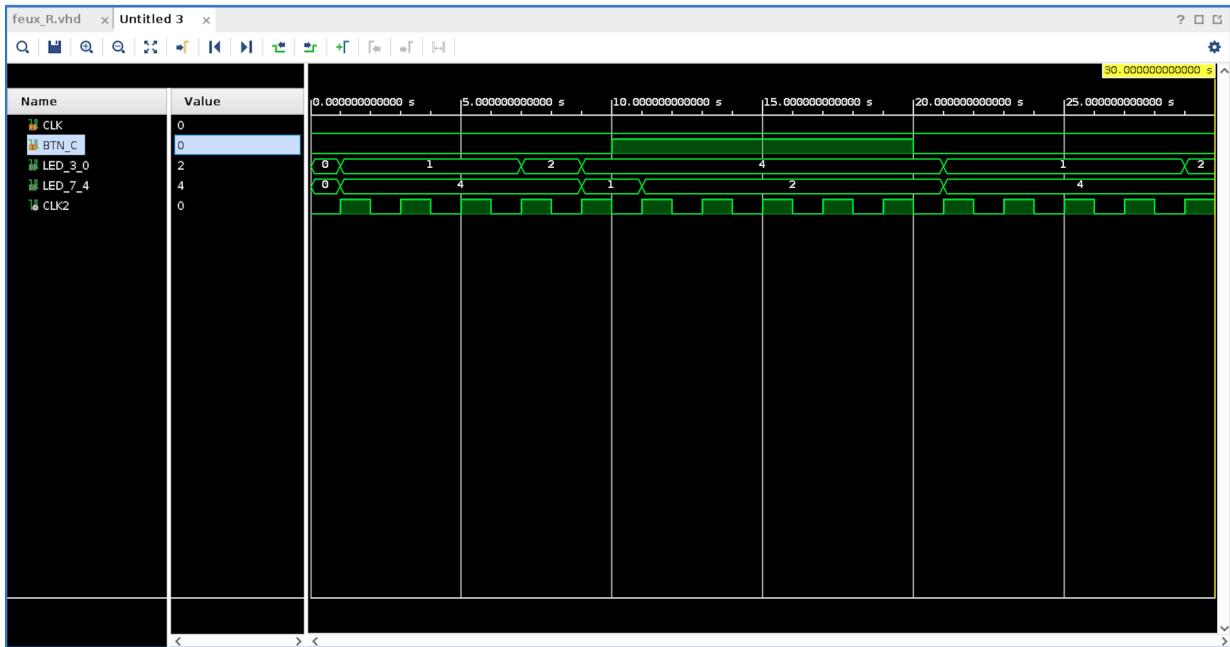


FIGURE 5 – *Simulation feux avec reset*

La simulation du circuit montre que le fonctionnement souhaité du contrôleur est bien représenté. En effet, à chaque top d'horloge, le système évolue de la même manière que le système sans reset. Cependant, lorsque que `BTN_C` (le reset) est activé, la machine à états retournera dans l'état correspondant à la remise à 0 en fonction de son état actuel.

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

### 3.1 Déroulement de la nouvelle séquence

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 pendant 10 secondes ; les feux de l'axe 1 sont verts pendant 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 restentverts jusqu'à détection de présence

### 3.2 Machine à état du feux avec capteur sans reset

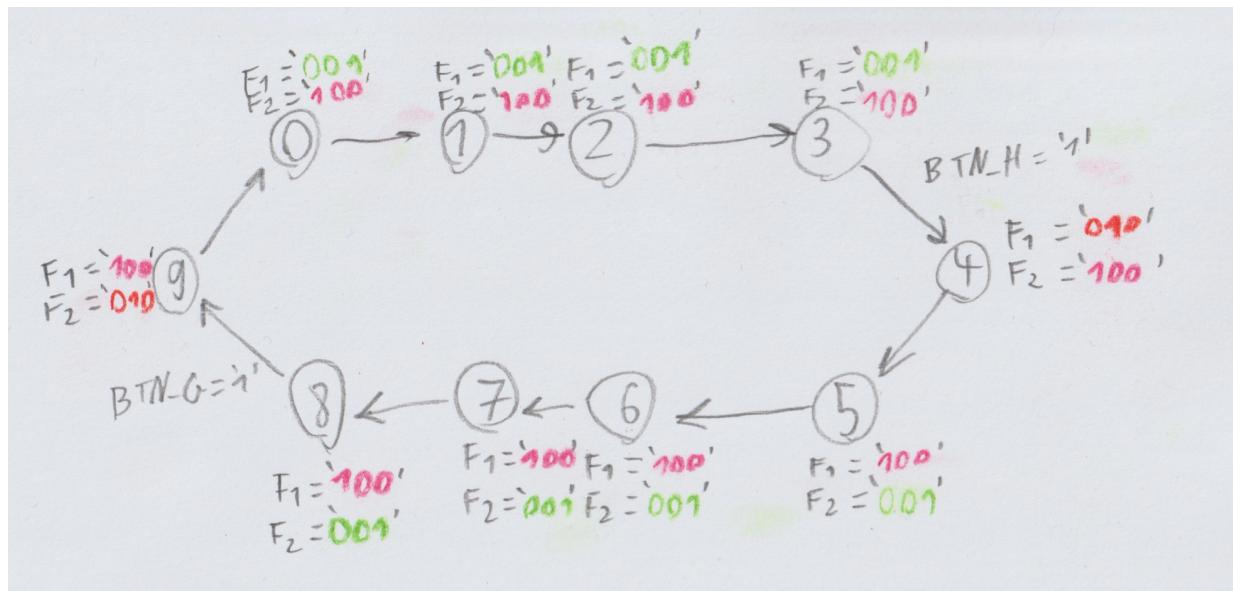


FIGURE 6 – Machine à états feux avec capteurs

### 3.3 Modélisation du système en VHDL

Nous avons écrit le code suivant dans le fichier feux\_C.vhd :

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity feux_C is
  Port(CLK,BTN_C, BTN_H,BTN_G, BTN_B : in bit;
        LED_3_0,LED_7_4 : out integer range 0 to 15);
end feux_C;

architecture Behavioral of feux_C is
  component clock_divider
    generic(divisor : integer);
    port(clock_in, reset : in bit;
          clock_out : out bit);
  end component clock_divider;
  Signal CLK2 : bit;

```

```

begin
    CD : clock_divider
        generic map(divisor => 200000000)
        port map(clock_in => CLK, reset => BTN_B, clock_out => CLK2);

process(CLK2)
    variable etat : integer range 0 to 11 := 11;
begin
    if(CLK2'event and CLK2='1') then
        case etat is
            when 0 => etat:=1;
            when 1 => etat:=2;
            when 2 => etat:=3;
            when 3 => if (BTN_H = '1') then etat := 4; else etat := 3; end if;
            when 4 => etat := 5;
            when 5 => etat:=6;
            when 6 => etat:=7;
            when 7 => etat:=8;
            when 8 => if (BTN_G = '1') then etat:= 9; else etat := 8; end if;
            when 9 => etat:=0;
            when others => etat:=0;
        end case;
        if (etat = 0) then
            LED_3_0 <= 1; LED_7_4 <= 4; end if;
        if (etat = 1) then
            LED_3_0 <= 1; LED_7_4 <= 4; end if;
        if (etat = 2) then
            LED_3_0 <= 1; LED_7_4 <= 4; end if;
        if (etat = 3) then
            LED_3_0 <= 1; LED_7_4 <= 4; end if;
        if (etat = 4) then
            LED_3_0 <= 2; LED_7_4 <= 4; end if;
        if (etat = 5) then
            LED_3_0 <= 4; LED_7_4 <= 1; end if;
        if (etat = 6) then
            LED_3_0 <= 4; LED_7_4 <= 1; end if;
        if (etat = 7) then
            LED_3_0 <= 4; LED_7_4 <= 1; end if;
        if (etat = 8) then
            LED_3_0 <= 4; LED_7_4 <= 1; end if;
        if (etat = 9) then
            LED_3_0 <= 4; LED_7_4 <= 2; end if;
        end if;
    end process;
end Behavioral;

```

### 3.4 Simulation avec capteurs

Nous obtenons la simulation suivante :

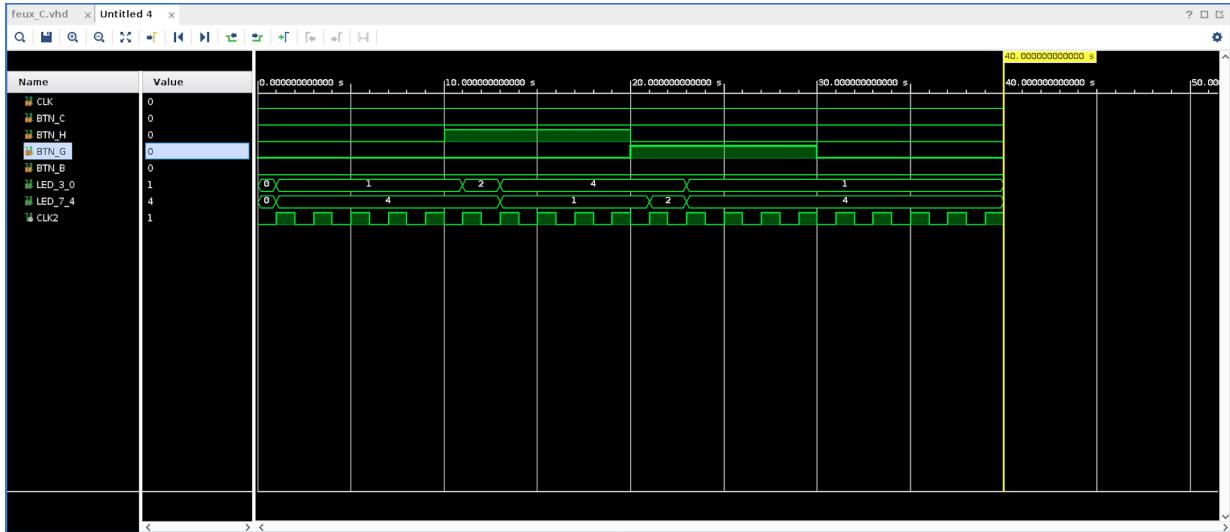


FIGURE 7 – *Simulation feux avec capteurs*

On peut voir à l'aide de cette simulation que dès lors qu'un capteur capte la présence d'une voiture, il déclenche après la fin de la période du feu vert en cours, la mise à l'orange puis rouge. Et inversement pour le feu et le capteur d'en face. De plus, nous pouvons observé que lorsque les capteurs ne captent aucune voiture dans un axe, alors l'axe opposé reste au vert. Comme par exemple LED\_3\_0 qui reste après le front descendant de BTN\_H jusqu'à la fin de la simulation.

### 3.5 Programmation du FPGA

Après génération du BitStream et programmation du FPGA, le système était fonctionnel. En effet, les boutons BTN\_H et BTN\_G, une fois pressés, renvoyaient le système à l'état attendu.

### 3.6 Feux avec capteur et reset synchrone

De la même manière que lors de l'exercice 2, nous devons assurer que lors d'une demande de reset, un feu vert ne passe pas au 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.

### 3.7 Machine à états du système

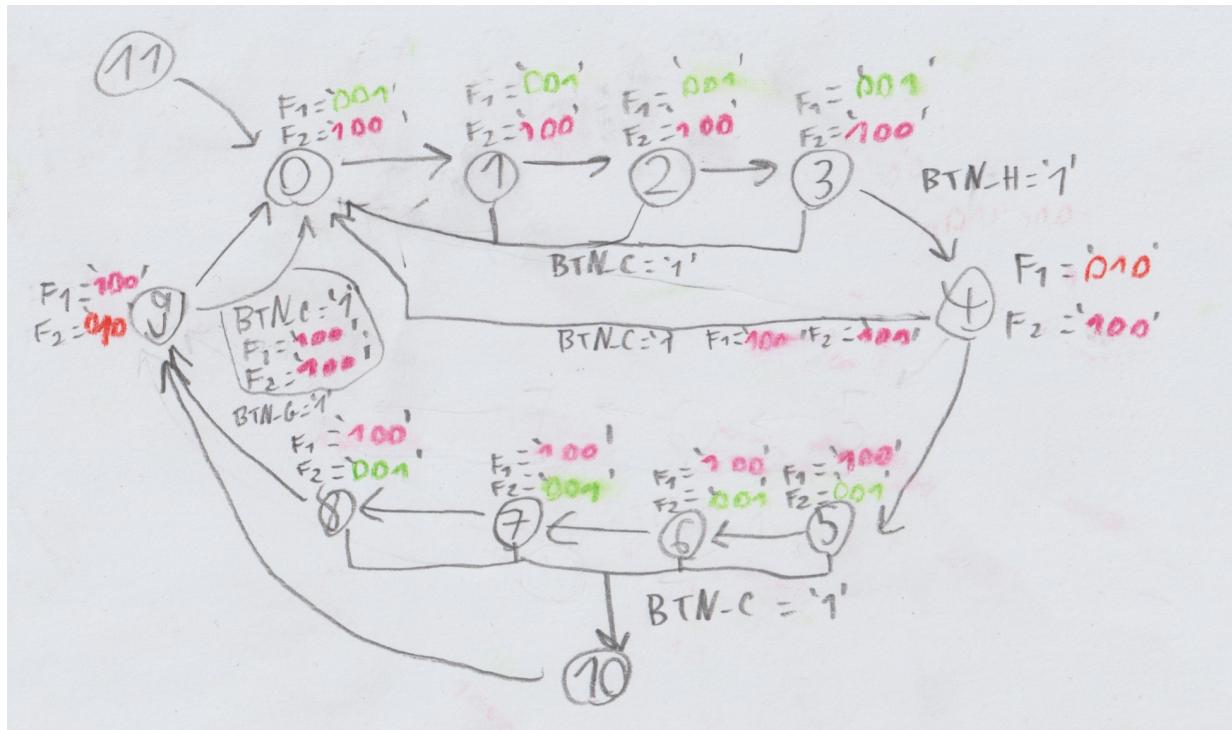


FIGURE 8 – Machine à états feux avec capteurs et reset synchrone

### 3.8 Modélisation du système en VHDL

Nous avons écrit le code suivant dans le fichier feux\_RC.vhd :

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity feux_RC is
    Port(CLK,BTN_C, BTN_H,BTN_G, BTN_B : in bit;
          LED_3_0,LED_7_4 : out integer range 0 to 15);
end feux_RC;

architecture Behavioral of feux_RC is
    component clock_divider
        generic(divisor : integer);
        port(clock_in, reset : in bit;
              clock_out : out bit);
    end component clock_divider;
    Signal CLK2 : bit;

begin
    CD : clock_divider
        generic map(divisor => 200000000)
        port map(clock_in => CLK, reset => BTN_B, clock_out => CLK2);

```

```

process(CLK2)
    variable etat : integer range 0 to 11 := 11;
begin
    if (BTN_C = '1') then
        if (etat = 0 or etat = 1 or etat = 2 or etat = 3) then
            etat := 0;
        end if;
        if (etat = 4) then
            LED_3_0 <= 4; LED_7_4 <= 4; etat := 0;
        end if;
        if (etat = 5 or etat = 6 or etat = 7 or etat = 8) then      etat := 10;
        end if;
        if (etat = 9) then LED_3_0 <= 4; LED_7_4 <= 4;
            etat := 0;
        end if;
    end if;
    if(CLK2'event and CLK2='1') then
        case etat is
            when 0 => etat:=1;
            when 1 => etat:=2;
            when 2 => etat:=3;
            when 3 => if (BTN_H = '1') then etat := 4; else etat := 3; end if;
            when 4 => etat := 5;
            when 5 => etat:=6;
            when 6 => etat:=7;
            when 7 => etat:=8;
            when 8 => if (BTN_G = '1') then etat:= 9; else etat := 8; end if;
            when 9 => etat:=0;
            when 10 => etat:=9;
            when others => etat:=0;
        end case;
        if (etat = 0) then
            LED_3_0 <= 1; LED_7_4 <= 4;
        end if;
        if (etat = 1) then
            LED_3_0 <= 1; LED_7_4 <= 4;
        end if;
        if (etat = 2) then
            LED_3_0 <= 1; LED_7_4 <= 4;
        end if;
        if (etat = 3) then
            LED_3_0 <= 1; LED_7_4 <= 4;
        end if;
        if (etat = 4) then
            LED_3_0 <= 2; LED_7_4 <= 4;
        end if;
        if (etat = 5) then
            LED_3_0 <= 4; LED_7_4 <= 1;
        end if;

```

```

        if (etat = 6) then
            LED_3_0 <= 4; LED_7_4 <= 1;
        end if;
        if (etat = 7) then
            LED_3_0 <= 4; LED_7_4 <= 1;
        end if;
        if (etat = 8) then
            LED_3_0 <= 4; LED_7_4 <= 1;
        end if;
        if (etat = 9) then
            LED_3_0 <= 4; LED_7_4 <= 2;
        end if;
    end if;
end process;
end Behavioral;

```

### 3.9 Simulation avec capteurs et reset synchrone

Nous avons obtenu la simulation suivante :

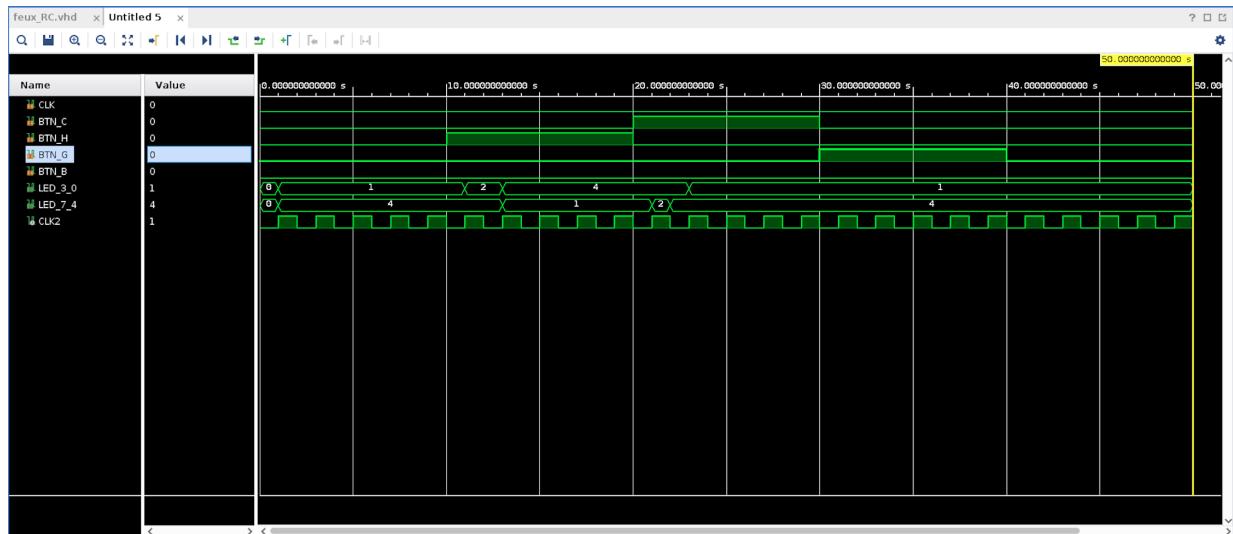


FIGURE 9 – *Simulation feux avec capteurs et reset*

On peut s'apercevoir que le système fonctionne parfaitement. En effet, lors du déclenchement du reset le système attend bien la fin de la période du vert pour renvoyer le feux à l'orange même si le capteur ne capte aucune voiture dans le sens opposé. De plus on peut également voir le passage au rouge du feux lorsque le capteur opposé capte la présence d'une voiture.

Après génération du BitStream et programmation du FPGA, le système était parfaitement fonctionnel. Malheureusement nous avons omis de prendre des photos du FPGA.

## 4 Conclusion

Dans ce TP, nous avons appris comment fonctionnait un `clock_divider` et comment s'en servir. Nous n'avons donc plus besoin d'appuyer sur un bouton pour modéliser l'horloge, elle s'exécute automatiquement. Lorsque nous n'avons pas besoin d'autres boutons comme pour la première partie de l'exercice 2, nous avons donc un système qui fonctionne tout seul, cycle après cycle, sans qu'on ait besoin d'intervenir.

Dans les exercices 2 et 3, nous avons pu découvrir 2 façons de programmer un système de feux. Nous avons remarqué que la deuxième façon était plus optimisée car elle permettait de fluidifier le trafic en évitant des feux rouges contre productifs.

De plus, l'étude d'un cas concret tel que le système de feux d'un carrefour nous a permis de réaliser l'importance du respect des contraintes lors de la réalisation d'un projet. En effet le respect des durées minimales des phases, et de l'ordre dans lesquelles elles peuvent se succéder (vert → orange → rouge → vert) était essentiel afin d'assurer un minimum de sécurité, notamment en ce qui concerne le reset du système.