Rapport TP 03

Samory DIABY

Table des matières

1 Objectif **2**

2 Notes **2**

3 Réalisations **2**

3.1 Nouveau compteur 2

3.1.1 Schéma RTL du nouveau compteur 2

3.1.2 Code VHDL du nouveau compteur 3

3.1.3 Testbench du nouveau compteur 5

3.2 Clignotement des LEDs 7

3.2.1 Schémas RTL 7

3.2.2 Entrées / sorties du système 8

3.2.3 Analyse de rapport de synthèse 9

3.2.4 Analyse de rapport de timing 9

3.3 Démo 11

1 Objectif

L'objectif de ce TP est de réaliser une architecture permettant de faire clignoter deux LEDs RGB en rouge, vert et bleu. Le pilotage des LEDs se fera à l'aide de machines à états.

2 Notes

Pour ce tp, j'ai assumé que les compteurs étaient codés sur 28 bits.

3 Réalisations

3.1 Nouveau compteur

3.1.1 Schéma RTL du nouveau compteur

Pour réaliser le circuit permettant de faire clignoter les 2 LEDs RGB de la carte j'ai d'abord dessiné le schéma RTL ci-dessous qui permet de compter le nombre de fois où la sortie du compteur **counter_unit** passe à '1'. Ce compteur (**counter_unit**) passe à '1' en fonction d'un paramètre générique donné en entrée à notre circuit, et va définir le temps des états allumé ou éteint des LEDs.

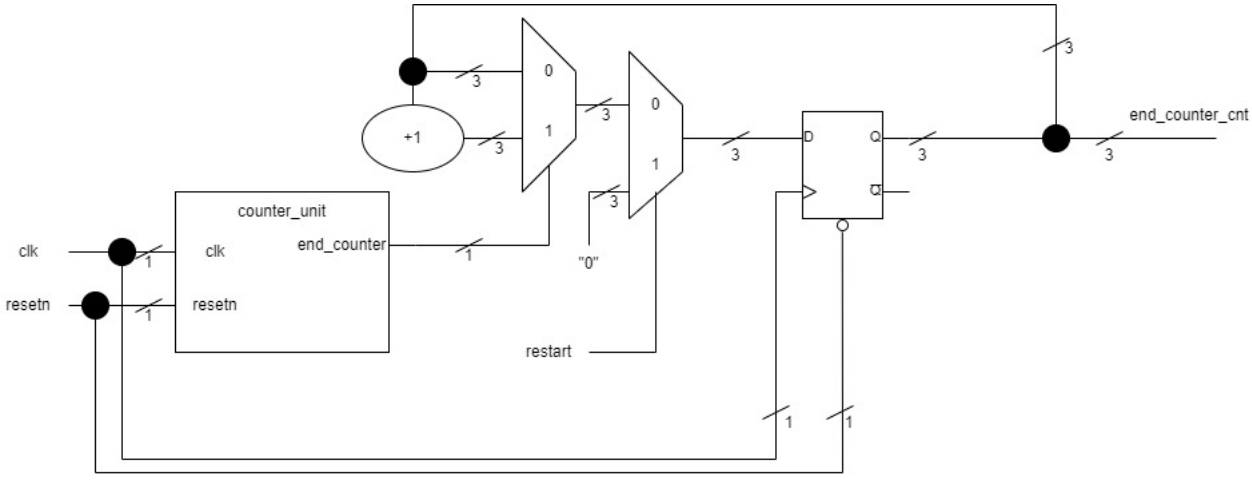

Figure 1 – Schéma RTL du compteur de "end_counter" (Q2)

3.1.2 Code VHDL du nouveau compteur

On retrouve ci-dessous le **code VHDL** associé au schéma précèdent :

```
entity tp_fsm is
    generic (
        --vous pouvez ajouter des parameres generics ici
        limit : unsigned(27 downto 0) := to_unsigned(8, 28) -- use in
           counter_unit
    );
    port (
    clk          : in std_logic;
        resetn         : in std_logic;
        restart          : in std_logic;
    end_counter_cnt : out std_logic_vector(27 downto 0)
     );
end tp_fsm;

architecture behavioral of tp_fsm is
    .
    .
    .

    -- counter
    -- intern_end_counter_cnt -> Q output of the flip flop in RTL
    signal intern_end_counter_cnt : std_logic_vector(27 downto 0) := (
       others => '0');

    -- muxes cmd signals
    signal cmd_cnt_incr : std_logic := '0'; -- will allow to choose to
       increment the counter or not
    signal cmd_cnt_restart : std_logic := '0'; -- will allow to choose to
       increment the counter or not

    -- counter_unit declaration
  component counter_unit
     generic(
         limit : unsigned(27 downto 0)
     );
     port (
         clk          : in std_logic;
         resetn       : in std_logic;
         end_counter   : out std_logic
     );
  end component;

  begin

        LED_COUNTER : counter_unit
        generic map (
           limit => limit -- set this value to change the time between LED
               blinks
        )
        port map (
            clk => clk,
```

```
47              resetn => resetn,
48              end_counter => cmd_cnt_incr
49          );
50
51
52      process(clk,resetn)
53      begin
54              if(resetn='1') then
55
56                  current_state <= idle;
57
58                  -- reset counter
59
60
61        elsif(rising_edge(clk)) then
62
63          current_state <= next_state;
64
65          --a completer avec votre compteur de cycles
66          -- counter incr mux
67          if cmd_cnt_incr = '1' then
68              intern_end_counter_cnt <= std_logic_vector(unsigned(
                    intern_end_counter_cnt) + 1);
69          -- else -> keep the prevoius value of the signal
70          end if;
71
72          -- counter restart mux
73          if cmd_cnt_restart = '1' then
74              intern_end_counter_cnt <= (others => '0');
75          -- else -> keep the prevoius value of the signal
76          end if;
77
78              end if;
79      end process;
80
81      .
82          .
83          .
84
85      -- combinatory logic
86      cmd_cnt_restart <= restart;
87      -- cmd_cnt_incr -> set by the counter_unit
88
89          -- output
90          end_counter_cnt <= intern_end_counter_cnt;
91
92  end behavioral;
```

3.1.3 Testbench du nouveau compteur

On retrouve ci-dessous le **testbench** associé au code précèdent :

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity tb_tp_fsm is
end tb_tp_fsm;

architecture behavioral of tb_tp_fsm is

  signal resetn          : std_logic := '0';
  signal clk             : std_logic := '0';
  signal restart         : std_logic := '0';
    signal end_counter_cnt : std_logic_vector(27 downto 0) := (others =>
       '0');

  -- Les constantes suivantes permette de definir la frequence de l'
     horloge
  constant hp        : time := 5 ns;  --demi periode de 5ns
  constant period    : time := 2*hp;  --periode de 10ns, soit une
     frequence de 100Hz


  component tp_fsm
     generic(
         limit : unsigned(27 downto 0)
     );
    port (
      clk         : in std_logic;
      resetn        : in std_logic;
      restart         : in std_logic;
        end_counter_cnt : out std_logic_vector(27 downto 0)
     );
  end component;

  begin
  dut: tp_fsm
      generic map (
         limit => to_unsigned(4, 28)
      )
        port map (
            clk => clk,
            resetn => resetn,
      restart => restart,
      end_counter_cnt => end_counter_cnt
        );

  --Simulation du signal d'horloge en continue
  process
    begin
    wait for hp;
    clk <= not clk;
```

```
49    end process;
50
51    process
52    begin
53
54      resetn <= '1';
55      wait for period*5;
56      resetn <= '0';
57
58      --a completer
59      wait;
60
61    end process;
62
63
64  end behavioral;
```

3.2 Clignotement des LEDs

3.2.1 Schémas RTL

Pour faire clignoter les LEDs, j'ai ajouté de la logique à mon schéma RTL précèdent (Par manque de place j'ai regroupé les sorties des LEDs RGB dans 2 groupes pour les LEDs 0 et 1) :

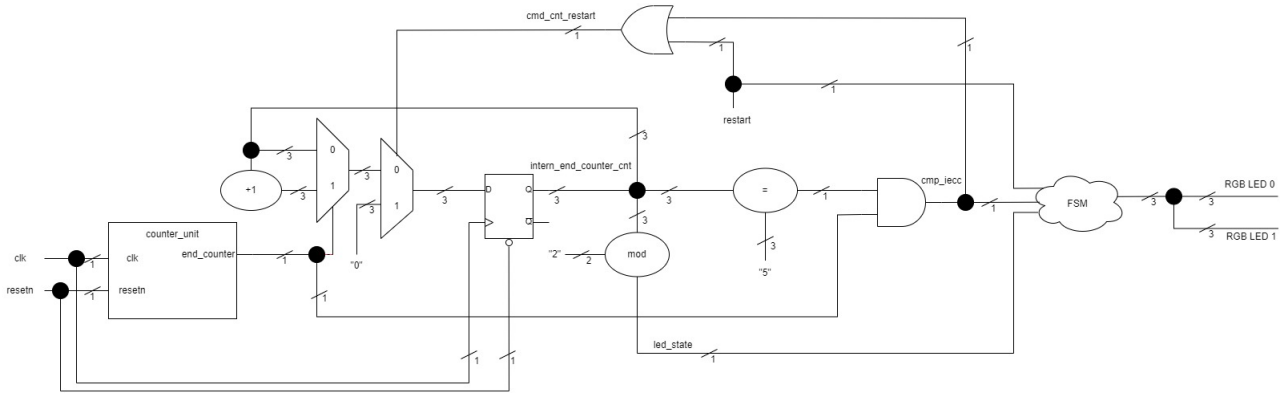

Figure 2 – Schéma RTL du circuit de clignotement des LEDs

Pour pouvoir faire clignoter les 2 LEDs RGB, j'utilise le résultat de comptage **intern_end_counter_cnt (iecc)** pour déterminer le nombre de clignotements. 1 Clignotement est équivalent à 1 cycle on/off des LEDs qui est représenté par 1 cycle tous les 2 iecc (1 demi-cycle à chaque incrément de iecc). Pour 1 état de LED, les 3 cycles de clignotements seront donc (premier cycle de clignotement en partant de 0) :

$$\text{cycle 1} \begin{cases} iecc = 0 & : & LED & off \\ iecc = 1 & : & LED & on \end{cases}$$

$$\text{cycle 2} \begin{cases} iecc = 2 & : & LED & off \\ iecc = 3 & : & LED & on \end{cases}$$

$$\text{cycle 3} \begin{cases} iecc = 4 & : & LED & off \\ iecc = 5 & : & LED & on \end{cases}$$

Pour gérer les cas **on** et **off** des LEDs, j'utilise la parité du nombre stocké dans iecc :

— Nombre impair -> on
— Nombre pair -> off

Pour ce faire j'utilise l'opérateur **modulo** avec en entrée le nombre à tester et le diviseur qui est ici **2**. Si l'opérateur renvoi **0** le nombre est pair, sinon il est impair. Ce résultat va alors mettre à jour l'état des LEDs dans la FSM via le signal interne **led_state** (les LEDs sont éteintes lorsque *led_state* est à '0' et allumées lorsque *led_state* est '1').
Le changement d'état dans la FSM se fait lorsque le compteur **iecc** a fini de compter. On compare iecc à 5 et on attend que **counter_unit** termine de compter son dernier cycle pour

éviter de changer d'état trop tôt. Autrement dès le passe à 5 de iecc, l'état changerait et on réinitialiserait le compteur au front montant suivant soit 10ns après le passage à 5.

Voici le schéma de la FSM servant à contrôler l'état des LEDs :

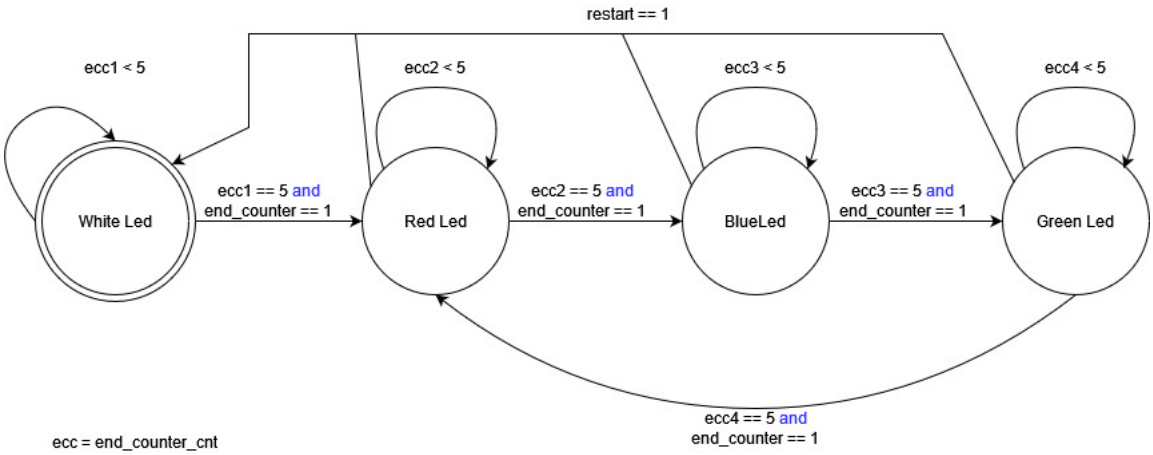

Figure 3 – Schéma RTL de la FSM de clignotement des LEDs

On peut voir sur le schéma RTL que l'on change d'état si :

— **iecc** à compté jusqu'à 5 et **counter_unit** vaut 1
— le bouton **restart** est pressé
— le bouton **resetn** est pressé

Dans le dernier cas, on reste dans l'état actuel.

3.2.2 Entrées / sorties du système

On a 9 entrées / sorties dans notre système qui sont :
— Entrées : clk, resetn, restart
— Sorties : **RGB LED 0** (led0_b, led0_g, led0_r) et **RGB LED 1** (led1_b, led1_g, led1_r)

3.2.3 Analyse de rapport de synthèse

En observant le rapport de synthèse, on retrouve notre machine à états :

State	New Encoding	Previous Encoding
white_led	00	00
red_led	01	01
blue_led	10	10
green_led	11	11

Figure 4 – FSM rapport de synthèse

On retrouve bien nos 4 états (LED blanche, rouge, verte et bleue) et il est indiqué que cette FSM est stockée dans un registre 2 bits nommé **FSM_sequential_current_state_reg**.

Dans le *schematic* on retrouve également la *counter_unit* qui est placé dans un module relié au reste du circuit. On a également le registre 28 bits **iecc** correspondant à notre compteur de *end_counter*.

3.2.4 Analyse de rapport de timing

On remarque sur le rapport de timing qu'il n'y a aucune violation (pas de violation de setup ni de hold) :

Design Timing Summary

WNS(ns)	TNS(ns)	TNS Failing Endpoints	TNS Total Endpoints	WHS(ns)	THS(ns)	THS Failing Endpoints	THS Total Endpoints	WPWS(ns)
4.030	0.000	0	86	0.244	0.000	0	86	4.500

Figure 5 – Rapport de timing

Chemin critique

D'après vivado, le chemin critique part du registre de comptage interne **iecc** et revient dans ce même registre. Cela correspondrait au circuit rouge sur le schéma ci-dessous :

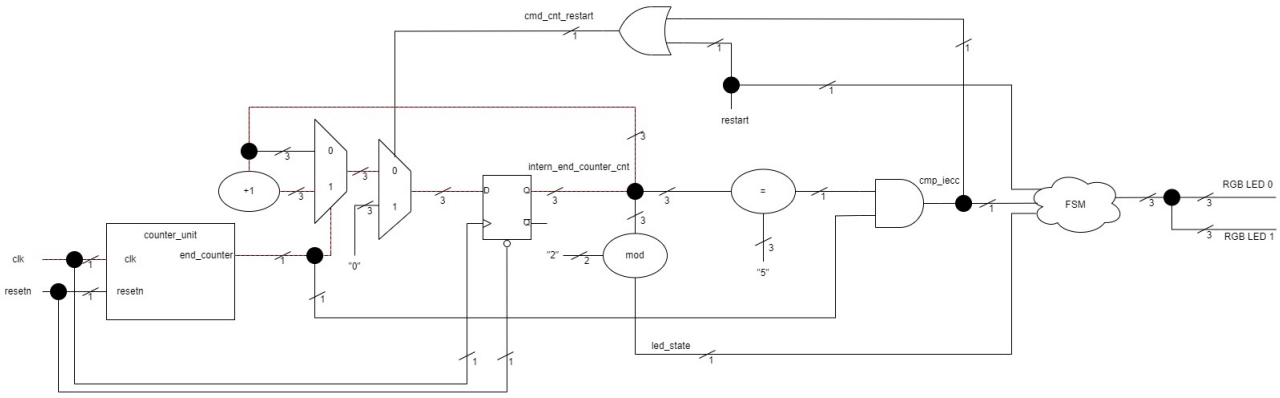

Figure 6 – Chemin critique

3.3 Démo

En observant les résultats de l'ILA de vivado, on peut valider le comportement de notre design. On va donc chercher à retrouver les différents états de notre FSM pour valider que les sorties sont correctes pour chaque état. On commence par l'état **White_led** :

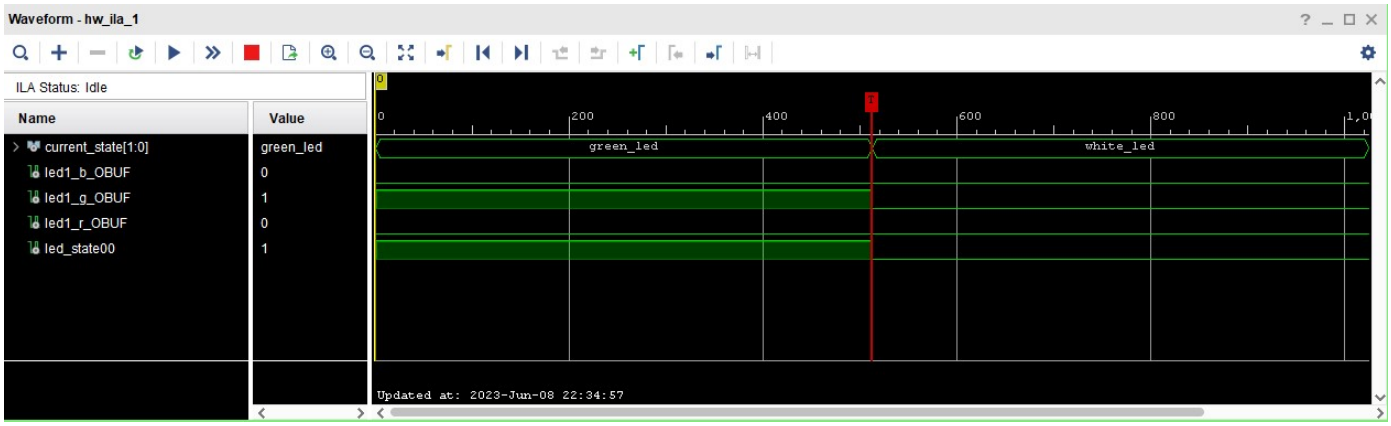

Figure 7 – ILA - LED blanche

Sur le chronogramme ci-dessus, on observe un passage à l'état **White_led** après un appui sur le **bouton 0** qui entraîne une remise à zéro du système et donc le retour à cet état (**White_led**). Les LEDs sont éteintes car le signal **led_state** est à '0' (La FSM était dans l'état **green_led** avec **led_state** à '1' quand le bouton a été appuyé).

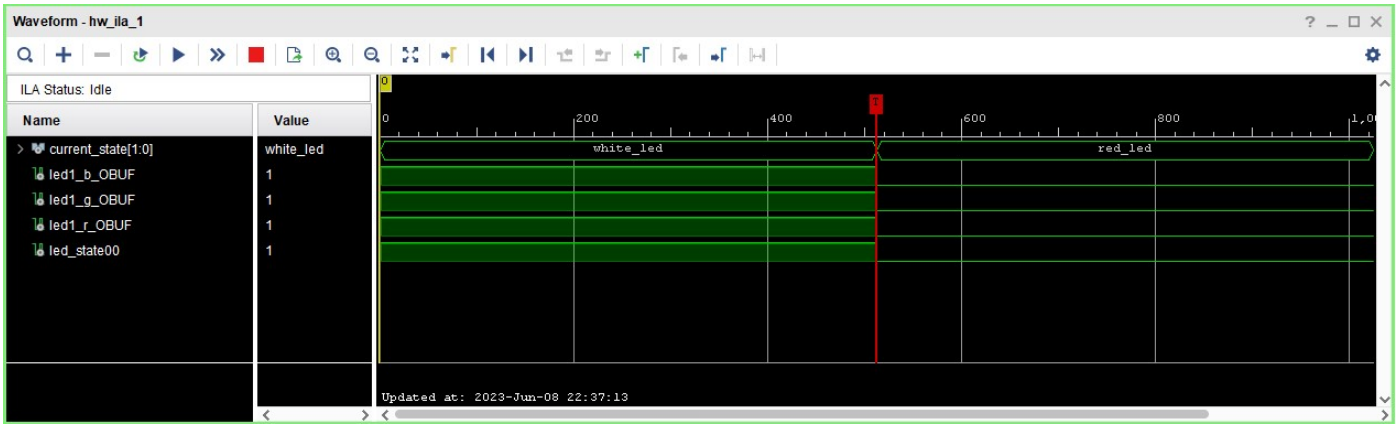

Figure 8 – ILA - LED rouge

Sur le chronogramme ci-dessus, on observe un passage à l'état **red_led** depuis l'état **white_led** après 3 cycles on/off des LEDs (observé visuellement). On remarque également que le dernier cycle de l'état **white_led** se termine bien avec l'ensemble des LEDs allumées (comme explicité en section 3.2.1).

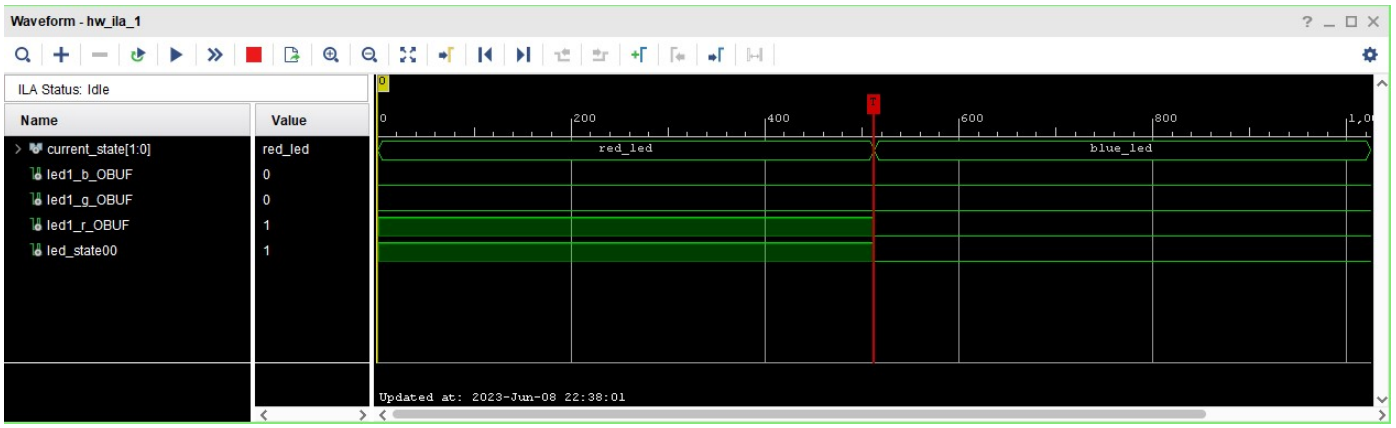

Figure 9 – ILA - LED bleue

Sur le chronogramme ci-dessus, on observe un passage à l'état **blue_led** depuis l'état **red_led** après 3 cycles on/off de la LED rouge (observé visuellement). On remarque également que le dernier cycle de l'état **red_led** se termine bien avec *uniquement* la LED rouge allumée (comme explicité en section 3.2.1).

Figure 10 – ILA - LED verte

Sur le chronogramme ci-dessus, on observe un passage à l'état **green_led** depuis l'état **blue_led** après 3 cycles on/off de la LED bleue (observé visuellement). On remarque également que le dernier cycle de l'état **blue_led** se termine bien avec *uniquement* la LED bleue allumée (comme explicité en section 3.2.1).