

Segment 2

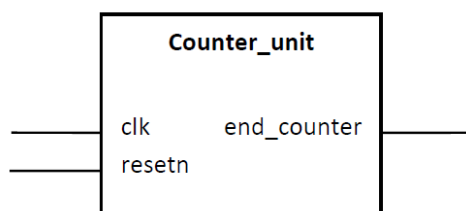
TP3 : FSM

Objectif

L'objectif de cet TP est faire clignoter une LED en utilisant un compteur de temporisation. Un compteur de temporisation permet de compter le nombre de coup d'horloge nécessaire pour attendre un temps voulu. En connaissant la fréquence de l'horloge il est possible de déterminer combien de périodes d'horloge il faut compter pour attendre 3 secondes par exemple.

Questions

1. Dans un fichier *.vhd*, créez un module *Counter_unit* à partir du compteur du TP1. Le module prendra en entrée un signal d'horloge et de resetn, et donnera en sortie le signal *end_counter*. Utilisez un paramètre *generic()* pour définir le nombre de coup d'horloge à compter. Le code de *Counter_unit* ne sera plus modifié ensuite.



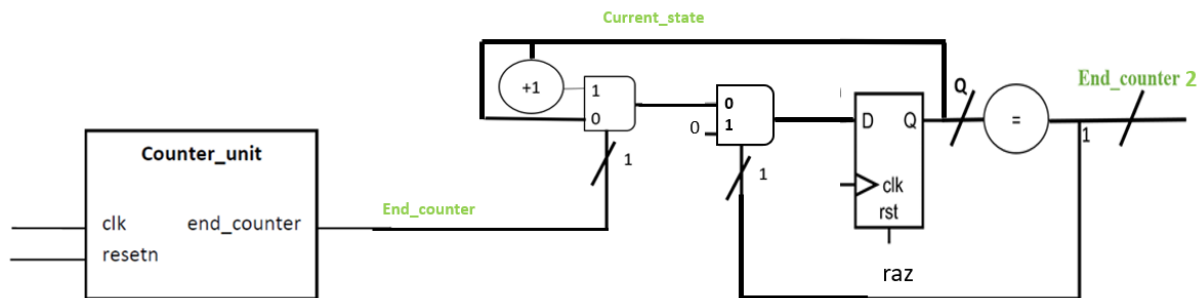
```

1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_unsigned.all;
4  use ieee.numeric_std.all;
5
6  entity counter_unit is
7  generic (
8      max_count : integer := 10;
9      nb_bit : integer := 4
10 );
11 port (
12     clk          : in std_logic;
13     resetn       : in std_logic;
14
15     end_counter  : out std_logic
16 );
17 end counter_unit;
18
19 architecture behavioral of counter_unit is
20
21     signal Q : std_logic_vector((nb_bit-1) downto 0);
22     signal end_count : std_logic;
23
24 begin
25     --Partie sequentielle
26     process(clk,resetn)
27     begin
28         if(resetn = '1') then
29             Q <= (others => '0');
30
31         elsif(rising_edge(clk)) then
32
33             Q <= Q + 1;
34             if(end_count = '1') then
35                 Q <= (others => '0');
36
37             end if;
38         end if;
39
40     end process;
41
42     --Partie combinatoire
43     end_count <= '1' when (Q = (max_count)-1)
44         else '0';
45
46     end_counter <= end_count;
47
48 end behavioral;

```

Dans ce module *Counter_unit*, nous avons utilisé un paramètre générique *max_count* pour définir le nombre de coups d'horloge à compter. Le compteur *counter_unit* est une valeur non signée de 4 bits qui compte jusqu'à *max_count* - 1. Lorsque le compteur atteint *max_count* - 1, le signal *end_counter* est mis à '1' pour indiquer que le comptage est terminé.

2. En schéma RTL, créez un compteur du signal *end_counter*. Ce compteur doit permettre de déterminer le nombre de cycles allumé/éteint qui ont été effectués par la LED. Le compteur doit pouvoir être remis à 0, maintenir sa valeur actuelle ou s'incrémenter.



Le passage à 1 de *end_counter 2* avec le paramètre générique permet de définir le temps des états des leds(allumé/éteint). Le nombre de cycles allumé/éteint dans notre tp est de 6. On a trois étapes ou cycle d'allumer et trois d'états d'éteints.

3. Ecrivez un code VHDL décrivant ce compteur de cycle, vous utiliserez le module *Counter_unit*.

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith;

entity tp_fsm is
    generic (
        max_count : integer :=4;
        nb_bit : integer := 3;
        nbre_cycle : positive :=6 --cycle de clignotement allumé et éteint
    );
    port (
        clk          : in std_logic;
        resetrn       : in std_logic;
        --a completer
        --restart      : in std_logic;
        end_counter2  : out std_logic;
        led_out_R     : out std_logic;
        led_out_B     : out std_logic;
        led_out_V     : out std_logic
    );
end tp_fsm;

architecture behavioral of tp_fsm is

    signal raz : std_logic;
    signal end_count : std_logic;
    signal end_count2 : positive :=0;

    --Declaration de l'entite a tester
    component counter_unit
        generic (
            max_count : integer :=4;
            nb_bit : integer := 3
        );
        port (
            clk          : in std_logic;
            resetrn       : in std_logic;

```

```

        end_counter    : out std_logic
    );
end component;

begin

compteur : counter_unit
generic map (
    max_count => 4,
    nb_bit => 3
)
port map (
    clk => clk,
    resetn => resetn,
    end_counter => end_count
);

-- Process séquentielle

process(clk,resetn)
begin

if(resetn='1') then
    end_count2<= 0;

    -- current_state <= idle;

elseif(rising_edge(clk)) then
    if(raz='0') then
        if (end_count='1') then
            end_count2<=end_count2 +1;
        elseif(end_count='0') then
            end_count2<=end_count2;
        end if;
    else end_count2<=0;

    --current_state <= next_state;
    end if;
end if;
    --a completer avec votre compteur de cycles

    raz <= '1' when (end_count2 = ((nbre_cycle)-1) and end_count = '1')
    else '0';
    end_counter2 <= raz;

end process;
end behavioral;

```

4. Tester votre architecture avec un testbench.

```

library ieee;
use ieee.std_logic_1164.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';
    --a completer
    signal end_count2 : positive :=0;
    signal end_counter2 : std_logic:= '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
    constant nbre_cycle : positive := 6 ;
    constant max_count : integer :=4;
    constant nb_bit : integer := 3;

    component tp_fsm
    port (
        clk          : in std_logic;
        resetn       : in std_logic;
        --a completer
        end_counter2 : out std_logic
    );
    end component;

begin
    dut: tp_fsm
    port map (
        clk => clk,
        resetn => resetn,
        --a completer
        end_counter2 => end_counter2
    );

    --Simulation du signal d'horloge en continue
    process

        --Simulation du signal d'horloge en continue
    process
    begin

        wait for hp;
        clk <= not clk;

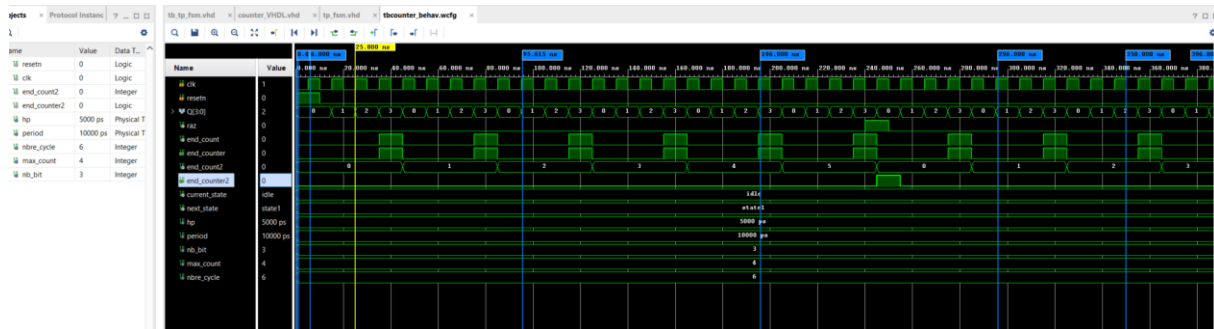
    end process;

    process
    begin

        resetn <= '1';
        wait for period*1;
        resetn <= '0';
        assert end_counter2='0'
        report "end_counter2 : test failed";
        wait for period*2;
        --a completer
        wait for period*nbre_cycle*max_count;
        assert end_counter2='1'
        report "end_counter2 : test failed";
        wait;
    end process;

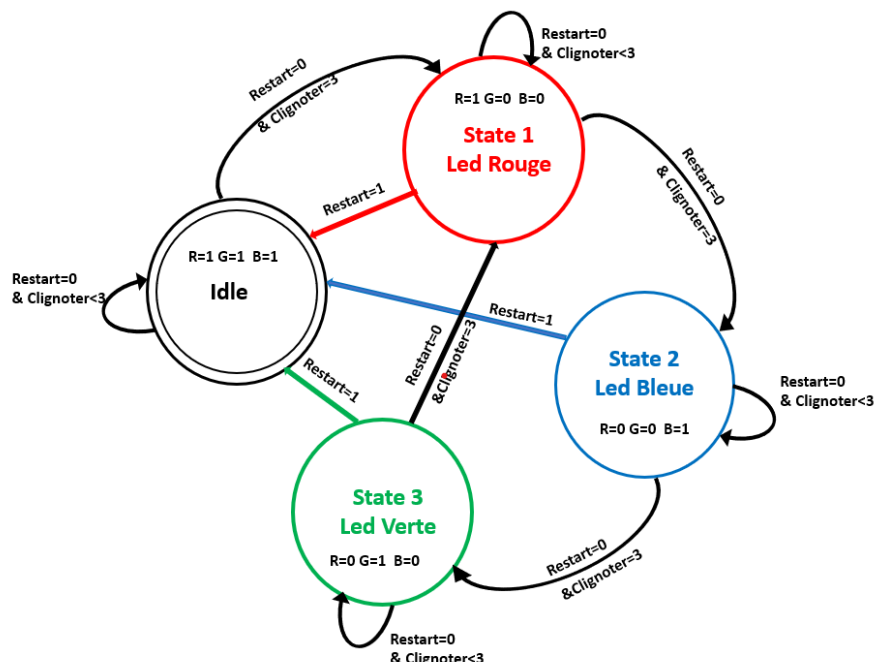
end behavioral;

```



On remarque que le comportement du compteur permet de déterminer le nombre de cycles allumé/éteint qui ont été effectués par la LED. Le compteur est ensuite remis à 0, pour maintenir sa valeur actuelle ou s'incrémenter. On compte 6 fronts montant d'end_counter et end_count2. End_couter2 est alors à 1, puis remise à 0 pour un nouveau cycle.

5. Créez en RTL une machine à états (FSM) permettant de faire clignoter une LED RGB en rouge puis bleu et enfin en vert avant de recommencer le cycle (rouge, bleu, vert, ...). Dans chaque état la LED devra clignoter 3 fois. De plus, si le bouton restart est appuyé, on retourne dans l'état initial quel que soit l'état dans lequel on se situe. L'état initial est l'état dans lequel on se situe au démarrage, on passe à l'état rouge après 3 clignotements de la LED en blanc (rouge, vert et bleu actifs en même temps).



6. Listez les signaux d'entrée, de sortie et les signaux internes de votre architecture.

Les signaux d'entrées

-Clk : l'horloge

-Resetn : le reset

-Restart : la remise à l'état initial de notre FSM

Les signaux de sorties

-end_counter2 : Sortie du compteur

-Led_out : Leds RGB

les signaux internes

current_state : état dans lequel se trouve la led actuellement

-next_state : état dans lequel va se trouver la led au prochain coup d'horloge

-s_led_out : signal interne pour led_out RGB

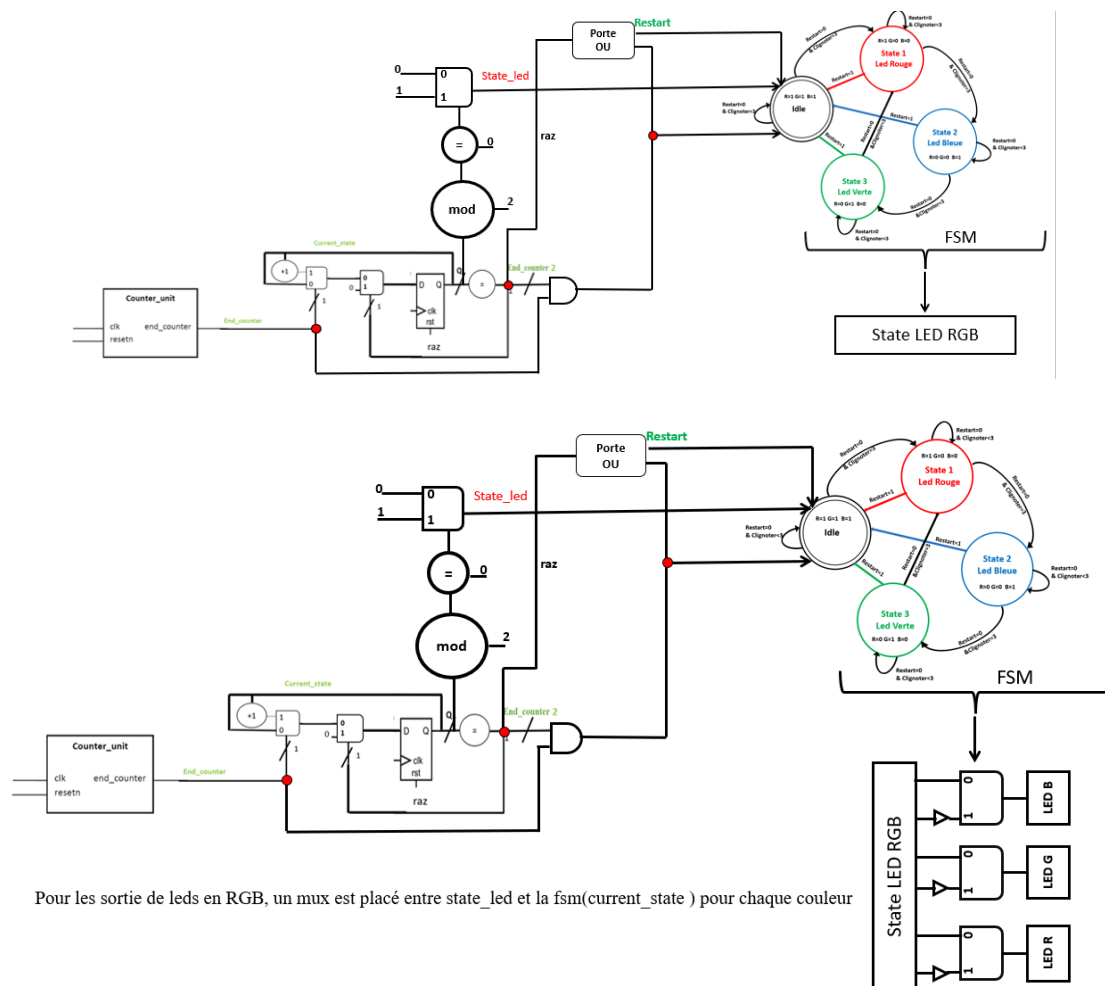
-count_clig : compte nombre de clignotement

-val_clig : permet la validation du current_state à next_state

-state_led : état de la led.

On garde tous les signaux internes déclarer pour le compteur unit et la remise à zéro.

7. Ajoutez à votre code VHDL les éléments que vous venez de créer.



Le schéma de l'ensemble, les signaux clock et resten sont reliés entres eux. Les points rouges sont les points de connexions du même signal. Sur ce schéma RTL, nous rajoutons à la FSM qui gère les états de la led, le module du compteur, des opérateurs logiques et le restart pour un bon fonctionnement. Pour gérer le clignotement de la led(éteint/allumé), un opérateur modulo et un diviseur 2 est utilisé. En sortie d'end_count2, nous avons 0 pour le pair ou 1 pour les impairs. Les leds sont allumées sur les impairs (state_led). La FSM vient ensuite gérer les états : initial, led rouge, led bleue et led verte. Le compteur changerait d'état après 6 cycles au front montant.

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_unsigned.all;
4  use ieee.std_logic_arith;
5
6  entity tp_fsm is
7  generic (
8      --max_count : integer :=200000000;
9      -- nb_bit : integer := 28;
10     max_count : integer :=4;
11     nb_bit : integer := 3;
12     nbre_cycle : positive :=6 --cycle de clignotement allumé et éteint
13 );
14 port (
15     clk          : in std_logic;
16     resetn       : in std_logic;
17     restart      : in std_logic;
18     end_counter2 : out std_logic;
19     led_out_R    : out std_logic;
20     led_out_B    : out std_logic;
21     led_out_V    : out std_logic
22 );
23 end tp_fsm;
24
25
26
27 architecture behavioral of tp_fsm is
28
29     type state is (idle, state1, state2, state3); --a modifier avec vos etats
30
31     signal current_state : state; --etat dans lequel on se trouve actuellement
32     signal next_state : state; --etat dans lequel on passera au prochain coup d'horloge
33     signal s_led_out_B : std_logic;
34     signal s_led_out_V : std_logic;
35     signal s_led_out_R : std_logic;
36     signal raz : std_logic;
37     signal end_count : std_logic;
38     signal end_count2 : integer range 0 to 5:=0;
39     signal count_clig : integer range 0 to 5:=0;
40     signal val_clig : std_logic;
41     signal state_led: std_logic;

```

```

--Declaration de l'entite a tester
component counter_unit
generic (
    max_count : integer :=20000000;
    nb_bit : integer := 28
);
port (
    clk : in std_logic;
    resetn : in std_logic;

    end_counter : out std_logic
);
end component;

begin

compteur : counter_unit
generic map (
    max_count =>20000000,
    nb_bit => 28
)
port map (
    clk => clk,
    resetn=>resetn,
    end_counter => end_count
);

-- Process séquentielle

process(clk,resetn,restart)
begin

if(resetn='1' or restart='1') then
    end_count2<= 0;
    current_state <= idle;

elseif(rising_edge(clk)) then
    current_state <= next_state;

    if(raz='0') then
        if (end_count='1') then
            end_count2<=end_count2 +1;
        elsif(end_count='0') then
            end_count2<=end_count2;
        end if;
    else
        end_count2<=0;
    end if;
    if (end_count2= nbre_cycle-1 and end_count='1') then
        count_clig <= count_clig + 1;
    elsif (val_clig='1') then
        count_clig<= 0;
    else
        count_clig<=count_clig;
    end if;
end if;

end process;

--Partie combinatoire a completer avec votre compteur de cycles
raz <= '1' when (end_count2 = ((nbre_cycle)-1) and end_count = '1')
    else '0';
end_counter2 <= raz;

-- check the state of end_counter2 to determine if the led should be 'on' or 'off'
state_led <= '0' when( end_count2 mod 2 = 0 )else '1';

```



```

115 -- FSM
116 process(current_state, restart, state_led, count_clig, end_counter2) --a completer avec vos signaux
117
118 begin
119     --signaux pilotes par la fsm
120
121     case current_state is
122     when idle =>
123         s_led_out_R<=state_led;
124         s_led_out_B<=state_led;
125         s_led_out_V<=state_led;
126         if restart='0' then
127
128             if(end_counter2='1') then
129                 next_state <= state1;
130
131             else
132                 next_state <= idle;
133                 end if;
134             --prochain etat
135         else
136             next_state<=idle;
137
138             end if;
139
140         when state1 =>
141             s_led_out_R<=state_led;
142             s_led_out_B<='0';
143             s_led_out_V<='0';
144             if restart='0' then
145
146                 if(end_counter2='1') then
147                     next_state <= state2;
148                     --prochain etat
149                 else
150                     next_state <= state1;
151
152                 end if;
153             else
154                 next_state <= idle;
155
156                 end if;
157
158         when state2 =>
159             s_led_out_R<='0';
160             s_led_out_B<=state_led;
161             s_led_out_V<='0';
162             if restart='0' then
163                 if(end_counter2='1') then
164                     next_state <= state3;
165                     --prochain etat
166                 else
167                     next_state <= current_state;
168                     end if;
169                 else
170                     next_state <= idle;
171                     end if;
172
173         when state3 =>
174             s_led_out_R<='0';
175             s_led_out_B<='0';
176             s_led_out_V<=state_led;
177             if restart='0' then
178                 if(end_counter2='1') then
179                     next_state <= state1;
180                     --prochain etat
181                 else
182                     next_state <= current_state;
183                     end if;
184                 else
185                     next_state <= idle;
186                     end if;
187
188             end case;
189
190         end process;
191
192     --signaux pilotes par la fsm
193     when others =>
194
195         next_state <= idle;
196
197     end case;
198
199
200 end process;
201
202 --Partie combinatoire a completer avec les leds
203 led_out_R<=s_led_out_R;
204 led_out_B<=s_led_out_B;
205 led_out_V<=s_led_out_V;
206
207 end behavioral;

```

8. Ecrivez un testbench pour tester votre architecture. Vérifiez à la simulation que vous obtenez le résultat attendu.

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_unsigned.all;
4  use ieee.std_logic_arith;
5
6
7  entity tb_tp_fsm is
8  end tb_tp_fsm;
9
10 architecture behavioral of tb_tp_fsm is
11
12     signal resetn      : std_logic := '0';
13     signal clk          : std_logic := '0';
14     signal restart      : std_logic := '0';
15     --a completer
16     signal end_count2   : integer range 0 to 5:=0;
17     signal end_counter2 : std_logic:= '0';
18     signal s_led_out_B  : std_logic;
19     signal s_led_out_V  : std_logic;
20     signal s_led_out_R  : std_logic;
21     signal count_clig : integer range 0 to 5:=0;
22     signal val_clig    : std_logic;
23     -- Les constantes suivantes permette de definir la frequence de l'horloge
24     constant hp : time := 5 ns;      --demi periode de 5ns
25     constant period : time := 2*hp;  --periode de 10ns, soit une frequence de 100Hz
26     constant nbre_cycle : positive := 6 ;
27     constant max_count : integer :=20000000;
28     constant nb_bit : integer := 28;
29     --constant max_count : integer :=4;
30     -- constant nb_bit : integer := 3;
31
32     --Déclaration du composant fsm
33     component tp_fsm
34     port (
35         clk          : in std_logic;
36         resetn       : in std_logic;
37         restart       : in std_logic;
38         --a completer
39         end_counter2 : out std_logic;
40         led_out_V    : out std_logic;
41         led_out_R    : out std_logic;
42         led_out_B    : out std_logic
43     );
44 end component;
45
46 begin
47     dut: tp_fsm
48     port map (
49         clk => clk,
50         resetn => resetn,
51         restart => restart,
52         --a completer
53
54         --a completer
55         end_counter2 => end_counter2,
56         led_out_B => s_led_out_B ,
57         led_out_V => s_led_out_V ,
58         led_out_R => s_led_out_R
59     );
60
61 --Simulation du signal d'horloge en continue
62 process
63 begin
64
65     wait for hp;
66     clk <= not clk;
67
68 end process;
69
70 --Simulation du signal resetn en continue avec affichage de end_counter2 qui fonctionne après le nombre de cycle
71 process
72 begin
73
74     resetn <= '1';
75     --wait for hp;
76     wait for period*1;
77     resetn <= '0';
78     assert end_counter2='0'
79     report "end_counter2 : test failed";
80     wait for period*2;
81     --a completer
82     wait for period*nbre_cycle*max_count;
83     assert end_counter2='1'
84     report "end_counter2 : test failed";
85     wait;
86 end process;
87
88 --simulation de leds et clignotement
89 process
90 begin
91     wait for period*1;
92     -- Restart
93     restart <= '1';
94     wait for 40 ns;
95     restart <= '0';
96     wait for period*nbre_cycle*max_count;
97
98     -- Verifions l'état initial idle (LED blanche)
99     assert (s_led_out_R = '1' and s_led_out_V = '1' and s_led_out_B = '1')
100     report "Initial state mismatch" severity error;
101
102     -- Wait for 3 clignotements en blanc (INITIAL state : idle)
103     wait for period*nbre_cycle*max_count;
104
105     -- Verifions RED statel
106     assert (s_led_out_R = '1' and s_led_out_V = '0' and s_led_out_B = '0')

```

```

-- Verifions RED state1
assert (s_led_out_R = '1' and s_led_out_V = '0' and s_led_out_B = '0')
report "RED state mismatch" severity error;

-- Wait for 3 clignotements en rouge (RED state)
wait for period*nbre_cycle*max_count;

-- Verifions BLUE state2
assert (s_led_out_R = '0' and s_led_out_V = '0' and s_led_out_B = '1')
report "BLUE state mismatch" severity error;

-- Wait for 3 clignotements en bleu (BLUE state)
wait for period*nbre_cycle*max_count;

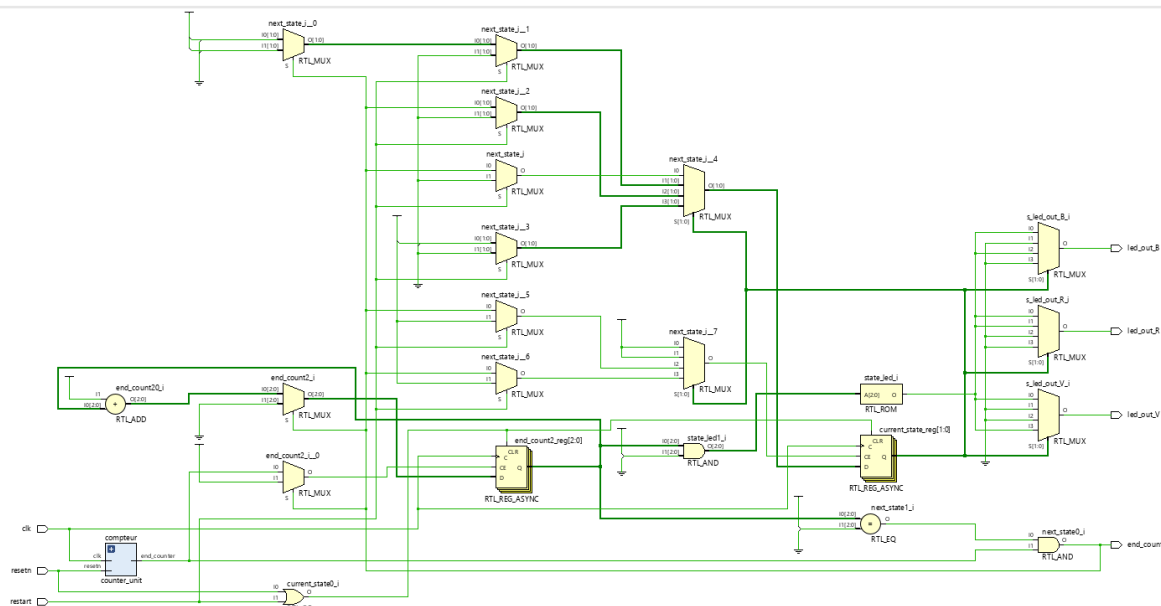
-- Verifions GREEN state3
assert (s_led_out_R = '0' and s_led_out_V = '1' and s_led_out_B = '0')
report "GREEN state mismatch" severity error;

-- Wait for 3 clignotements en vert (GREEN state)
wait for period*nbre_cycle*max_count;

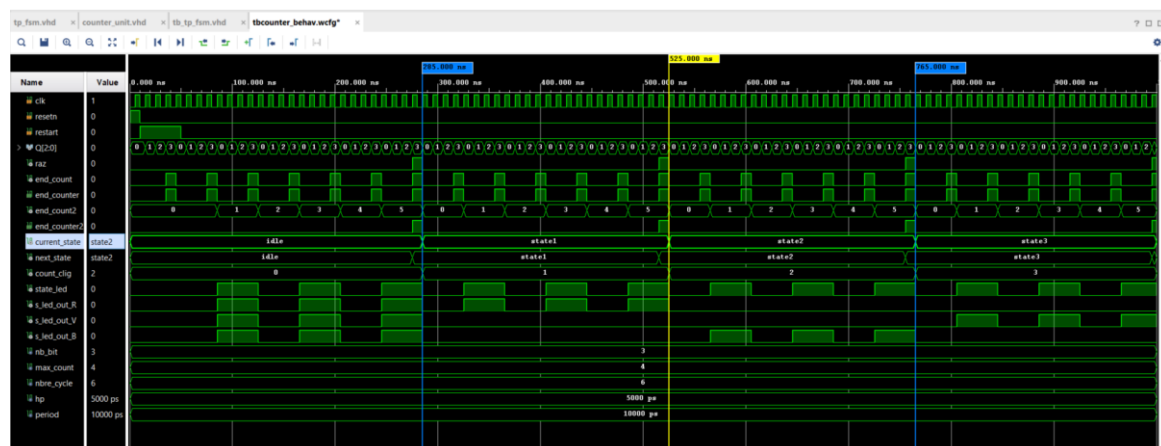
--Pour finir la simulation
wait;
end process;
end behavioral;

```

Le RTL de l'ensemble



Les résultats de simulations sur le chronogramme montre que cela fonctionne bien.



9. Exécutez la synthèse et relevez les ressources utilisées (y compris la FSM). Sur la schématique, identifiez où se situe votre compteur de cycle.

Nous retrouvons dans la synthèse les états de nos leds. Ils sont stockés dans un registre de 2 bits.

```
INFO: [Synth 8-802] inferred FSM for state register 'current_state_reg' in module 'tp_fsm'

-----
State |                New Encoding |                Previous Encoding
-----
idle  |                00 |                00
state1 |                01 |                01
state2 |                10 |                10
state3 |                11 |                11
-----

INFO: [Synth 8-3354] encoded FSM with state register 'current_state_reg' using encoding 'sequential' in module 'tp_fsm'
```

La description du RTL dans la synthèse.

```
Start RTL Component Statistics
-----
Detailed RTL Component Info :
+---Adders :
      2 Input   3 Bit   Adders := 2
+---Registers :
      3 Bit   Registers := 2
+---Muxes :
      2 Input   3 Bit   Muxes := 2
      4 Input   2 Bit   Muxes := 1
      2 Input   2 Bit   Muxes := 6
      2 Input   1 Bit   Muxes := 2
      4 Input   1 Bit   Muxes := 4
-----
Finished RTL Component Statistics

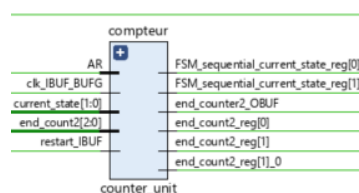
Start Writing Synthesis Report
-----
Report BlackBoxes:
+-----+
| BlackBox name | Instances |
+-----+
+-----+
Report Cell Usage:
+-----+
| Cell | Count |
+-----+
|1| BUFG | 1|
|2| LUT1 | 1|
|3| LUT2 | 4|
|4| LUT3 | 2|
|5| LUT4 | 2|
|6| LUT5 | 3|
|7| FDCE | 7|
|8| IBUF | 3|
|9| OBUF | 4|
```

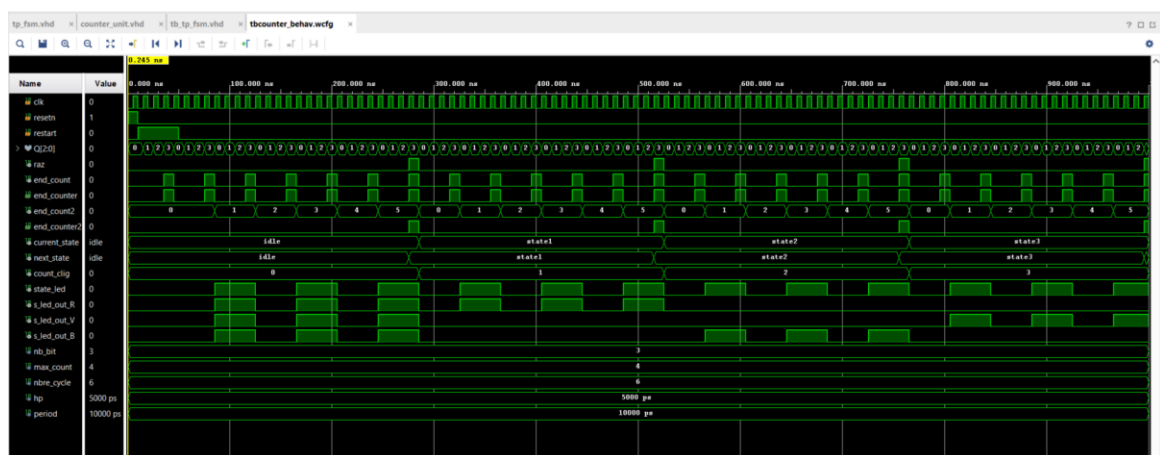
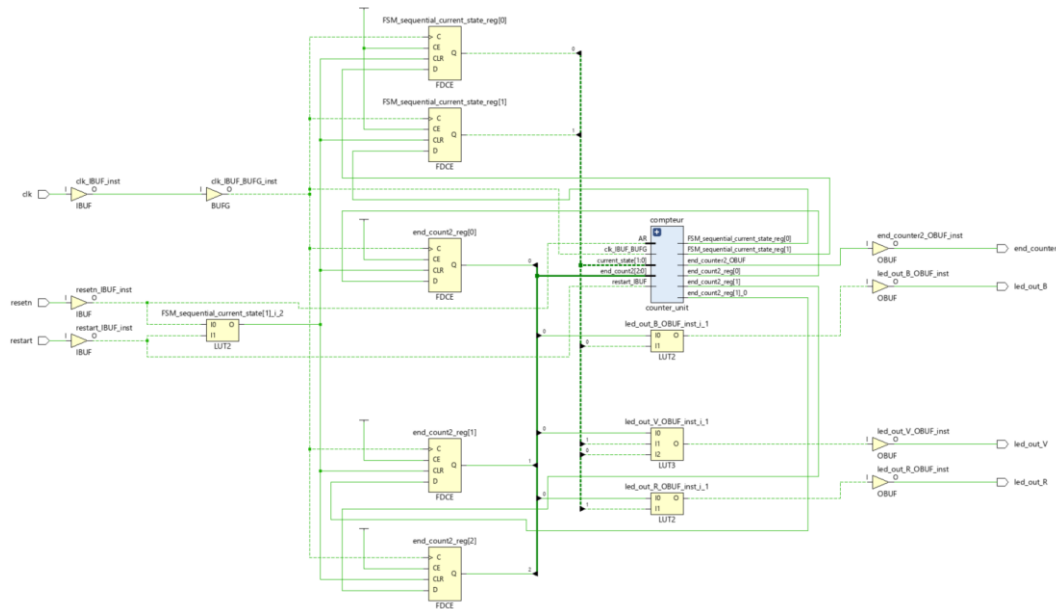
Tous les ressources sont répertoriées ci-dessus. Pour un générique de 4, on a 7 registres (deux pour la machine d'état, 3 pour le counter2 et 2 pour le counter_unit).

Lorsque le générique est de 200000000, on adonc 33 registres (deux pour la machine d'état, 3 pour le counter2 et 28 pour le counter_unit).

```
-----
Start RTL Component Statistics
-----
Detailed RTL Component Info :
+---Adders :
      2 Input   3 Bit   Adders := 1
+---Registers :
      3 Bit   Registers := 1
+---Muxes :
      2 Input   3 Bit   Muxes := 1
      4 Input   2 Bit   Muxes := 1
      2 Input   2 Bit   Muxes := 6
      2 Input   1 Bit   Muxes := 2
      4 Input   1 Bit   Muxes := 4
-----
Report Cell Usage:
+-----+
| Cell | Count |
+-----+
|1| BUFG | 1|
|2| CARRY4 | 7|
|3| LUT2 | 33|
|4| LUT3 | 2|
|5| LUT4 | 7|
|6| LUT5 | 2|
|7| LUT6 | 3|
|8| FDCE | 33|
|9| IBUF | 3|
|10| OBUF | 4|
```

Dans la synthèse : voici le compteur de cycle. On retrouve autour de lui, les registres qui gère le clignotement et la FSM.





10. Modifiez le fichier de contraintes pour connecter vos entrées / sorties du système avec les broches de la carte. Réglez l'horloge pour que sa fréquence soit à 100MHz.

```

1  ## This file is a general .xdo for the Cora E7-07S Rev. B
2  ## To use it in a project:
3  ## - uncomment the lines corresponding to used pins
4  ## - rename the used ports (in each line, after get_ports) according to the top level signal names in the project
5
6  # PL System Clock
7  set_property -dict {PACKAGE_PIN H16 IOSTANDARD LVCMOS33} [get_ports clk]
8  create_clock -period 10.000 -name sys_clk_pin -waveform (0.000 5.000) -add [get_ports clk]
9
10 # RGB LEDs
11 set_property -dict {PACKAGE_PIN L15 IOSTANDARD LVCMOS33} [get_ports {led_out_R}]
12 set_property -dict {PACKAGE_PIN G17 IOSTANDARD LVCMOS33} [get_ports {led_out_B}]
13 set_property -dict {PACKAGE_PIN N15 IOSTANDARD LVCMOS33} [get_ports {led_out_V}]
14 set_property -dict { PACKAGE_PIN G14 IOSTANDARD LVCMOS33 } [get_ports { led1_b }]; #IO_0_35 Sch=led1_b
15 set_property -dict { PACKAGE_PIN L14 IOSTANDARD LVCMOS33 } [get_ports { end_counter2 }]; #IO_L22P_T3_AD7P_35 Sch=led1_g
16 set_property -dict { PACKAGE_PIN M15 IOSTANDARD LVCMOS33 } [get_ports { led1_r }]; #IO_L23N_T3_35 Sch=led1_r
17
18 # Buttons
19 set_property -dict { PACKAGE_PIN D20 IOSTANDARD LVCMOS33 } [get_ports { resetn }]; #IO_L4N_T0_35 Sch=btn[0]
20 set_property -dict {PACKAGE_PIN D19 IOSTANDARD LVCMOS33} [get_ports {restart}]

```

La fréquence est réglée à 100MHZ. La schématique est représentée ci-dessous.

11. Lancez l'implémentation puis étudiez le rapport de timing (vérifiez les violations de set up et de hold et identifiez le chemin critique).

Le Clock

Clock Summary			

Clock	Waveform(ns)	Period(ns)	Frequency(MHz)

dbg_hub/inst/BSCANID.u_xsdm_id/SWITCH_N_EXT_BSCAN.bscan_inst/SERIES7_BSCAN.bscan_inst/TCK	{0.000 16.500}	33.000	30.303
sys_clk_pin	{0.000 5.000}	10.000	100.000

On vérifie bien que la période est à 10ns et la fréquence est de 100MHz

Les valeurs dans le THS et TNS sont à 0, il n'y a pas de violation du set up et du hold. Pas de métastabilité.

Design Timing Summary											

WNS(ns)	TNS(ns)	TNS Failing Endpoints	TNS Total Endpoints	WHS(ns)	THS(ns)	THS Failing Endpoints	THS Total Endpoints	WPS(ns)	TPWS(ns)	TPWS Failing Endpoints	TPWS Total Endpoints
4.026	0.000	0	3929	0.023	0.000	0	3913	3.750	0.000	0	2200

All user specified timing constraints are met.

Le chemin critique est :

Max Delay Paths

Slack (MET) : 25.927ns (required time - arrival time)

Source:

dbg_hub/inst/BSCANID.u_xsdm_id/SWITCH_N_EXT_BSCAN.bscan_switch/state_reg[0]/C

(rising edge-triggered cell FDRE clocked by
dbg_hub/inst/BSCANID.u_xsdm_id/SWITCH_N_EXT_BSCAN.bscan_inst/SERIES7_BSCAN.bscan_inst/TCK {rise@0.000ns fall@16.500ns period=33.000ns})

Destination:

dbg_hub/inst/BSCANID.u_xsdm_id/SWITCH_N_EXT_BSCAN.bscan_switch/portno_temp_reg[3]/D

(rising edge-triggered cell FDRE clocked by
dbg_hub/inst/BSCANID.u_xsdm_id/SWITCH_N_EXT_BSCAN.bscan_inst/SERIES7_BSCAN.bscan_inst/TCK {rise@0.000ns fall@16.500ns period=33.000ns})

Max Delay Paths	

Slack (MET) :	25.927ns (required time - arrival time)
Source:	dbg_hub/inst/BSCANID.u_xsdm_id/SWITCH_N_EXT_BSCAN.bscan_switch/state_reg[0]/C (rising edge-triggered cell FDRE clocked by dbg_hub/inst/BSCANID.u_xsdm_id/SWITCH_N_EXT_BSCAN.bscan_inst/SERIES7_BSCAN.bscan_inst/TCK {rise@0.000ns fall@16.500ns period=33.000ns})
Destination:	dbg_hub/inst/BSCANID.u_xsdm_id/SWITCH_N_EXT_BSCAN.bscan_switch/portno_temp_reg[3]/D (rising edge-triggered cell FDRE clocked by dbg_hub/inst/BSCANID.u_xsdm_id/SWITCH_N_EXT_BSCAN.bscan_inst/SERIES7_BSCAN.bscan_inst/TCK {rise@0.000ns fall@16.500ns period=33.000ns})
Path Group:	dbg_hub/inst/BSCANID.u_xsdm_id/SWITCH_N_EXT_BSCAN.bscan_inst/SERIES7_BSCAN.bscan_inst/TCK
Path Name:	Report Max at Slow Process Connect

12. Générez le bitstream pour vérifier le système sur carte.

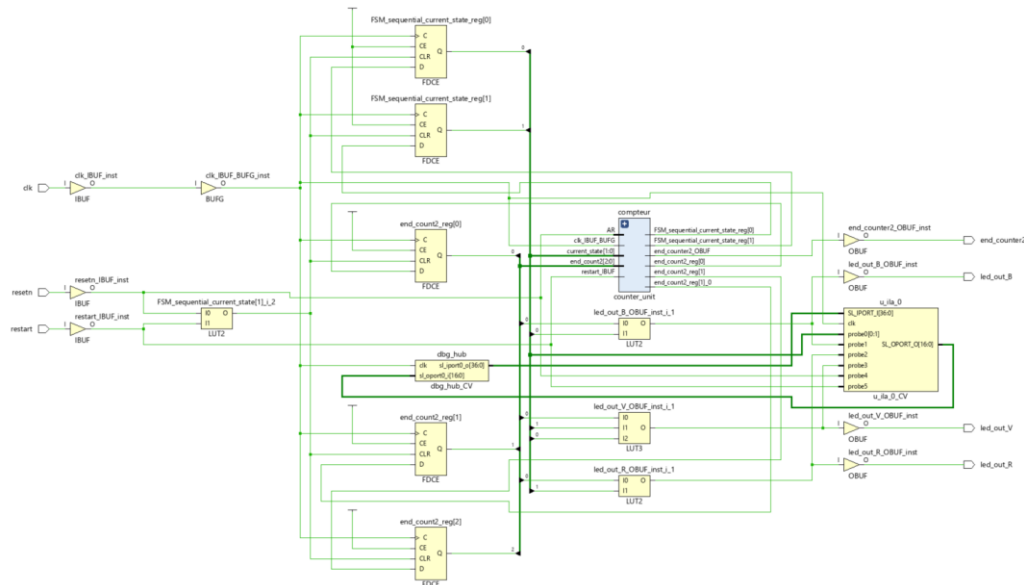
Les signaux leds out , et current_state sont observés dans le trigger.

hw_ila_1

Pour rappel, la principale fonction d'une ILA est de surveiller et d'analyser les signaux numériques à l'intérieur d'un circuit intégré ou d'un FPGA (Field-Programmable Gate Array). Elle permet de détecter les erreurs, de valider le fonctionnement des circuits et de comprendre

le comportement des signaux lors de l'exécution d'un programme ou d'une séquence d'opérations.

Nous démarrons l'enregistrement de l'ILA sur le front montant et descendant du port de sortie du current_state. On procède à un changement de valeur sur le signal de déclenchement entraînera l'ILA pour commencer à enregistrer les signaux sondés (Led,out RGB). Ceci est fait dans le déclencheur (trigger setup).



ILA Core Properties

hw_ila_1

Name: hw_ila_1

Cell: u_ila_0

Device: xc7z010_1

HW core: core_2

Capture sample count: 512 of 1024

Core status: Waiting For Trigger

Settings - hw_ila_1 × Status - hw_ila_1

Trigger mode: BASIC_ONLY

Capture Mode Settings

Capture mode: ALWAYS

Number of windows: 1 [1 - 1024]

Window data depth: 1024 [1 - 1024]

Trigger position in window: 512 [0 - 1023]

General Settings

Refresh rate: 10000 ms

Trigger Setup - hw_ila_1 × Capture Setup - hw_ila_1

Name	Operator	Radix	Value	Port	Comparator Usage
led_out_B_OBUF	==	[B]	B	probe1[0]	1 of 1
current_state[1:0]	==	[H]	X	probe0[1:0]	
led_out_R_OBUF	==	[B]	B	probe2[0]	1 of 1
led_out_V_OBUF	==	[B]	B	probe3[0]	1 of 1

Nous voyons la ligne verticale rouge (marqueur) sur le front montant de notre signal de déclenchement (port trigger de led), et il est en position 512. Nous pouvons également vérifier que le signal compte se comporte correctement et change de couleur suivant les changements d'état.

