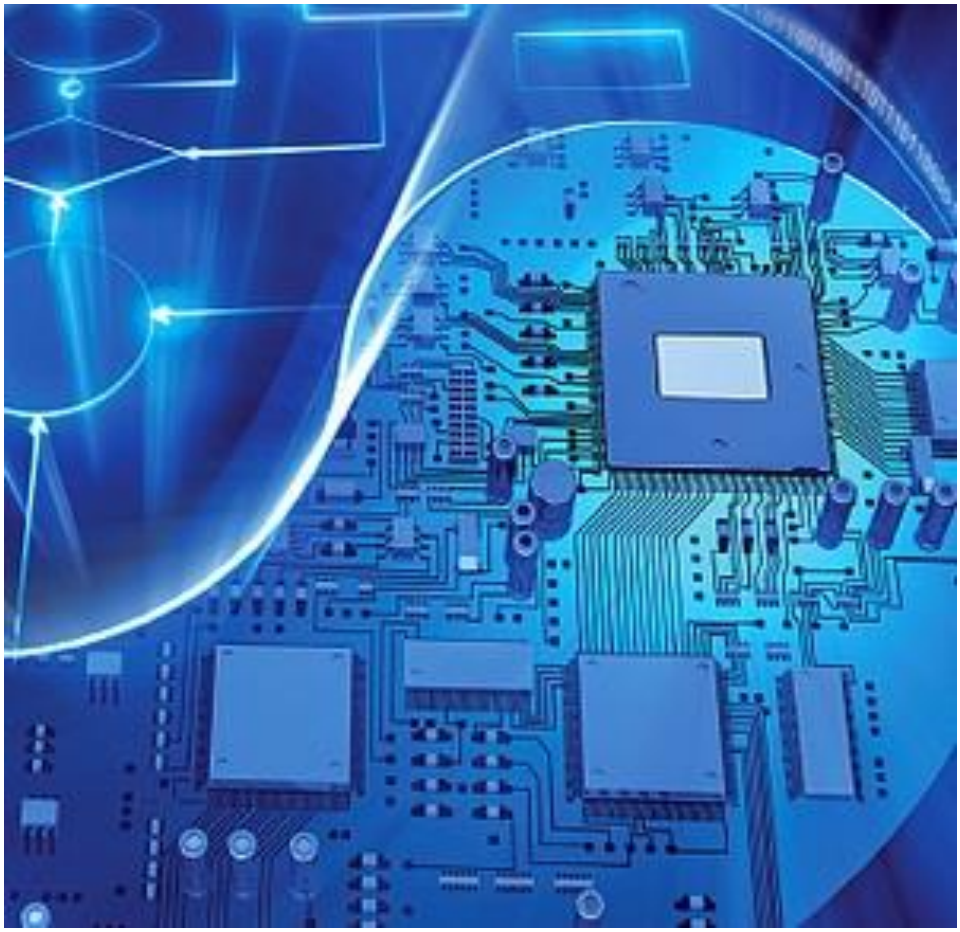


CSN

Labo 06

Par

A Gabriel Catel Torres
Pierrick Muller

Professeur : Mr Messerli
Assistant : Sebastien Masle

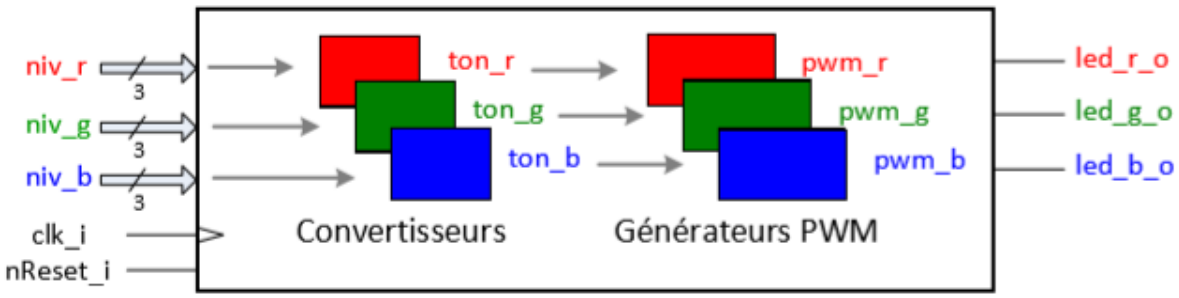
Table des matières

Introduction

L'objectif de ce laboratoire est de réaliser une commande d'une LED-RGB permettant de contrôler l'intensité et la couleur via des signaux binaires. Le principe du PWM sera utilisé pour contrôler l'intensité.

Un PWM permet de contrôler via un signal binaire une information analogique. La modulation de largeur d'impulsions (PWM) est une technique couramment utilisée pour synthétiser des signaux continus à l'aide d'un signal tout ou rien numérique ('1' et '0'). La variation de la tension continue est obtenue en modifiant le rapport cyclique entre la durée à '1' et à '0' du signal digital.

Le système comprend 2 blocs, soit les convertisseurs et les PWM. L'intensité des leds n'étant pas linéaire en fonction de ton, nous allons utiliser des convertisseurs pour obtenir 8 niveaux d'intensité régulièrement répartit. Le second bloc comprend les 3 générateurs PWM, soit un pour chaque couleur. Voici le schéma bloc du système :



1. Réalisation des générateurs PWM

1.1 Analyse du fonctionnement des générateurs PWM

- Une période PWM (sa fréquence) qui est de 1.024 ms. Nous avons une fréquence système de 1MHz. Ainsi pour obtenir la bonne période il nous faut utiliser un compteur 10 bits. Avec 10 bits il est possible de représenter exactement 1024 valeurs (0 .. 1023), donc les 1024 coups de clocks, ainsi nous aurons un compteur qui tournera constamment et qui n'aura pas besoin de comparateur puisque la valeur maximale est la valeur de la fin de période.

- Des signaux qui arrivent dans le bloc. Ces signaux sont ton_r, ton_g et ton_b. Ces valeurs sont sur 8 bits et vont donner une valeur de sortie des trois couleurs qui gère la led. Cette valeur détermine un temps où le signal sera à 1 par rapport à la période. Avec un calcul, le système derrière va déterminer l'intensité que l'on a demandé sur chaque led pour donner la bonne couleur à cette dernière.

- Le compteur que nous allons utiliser est sur 10 bits. C'est la valeur de sortie de celui-ci que nous allons utiliser pour comparer les valeurs des ton_* précédemment expliquées. Nous aurons dans cette partie trois comparateurs, un pour chaque couleur. En entrée du comparateur on branche la valeur du compteur à laquelle nous comparons la valeur des ton_*. La valeur de sortie de chaque comparateur (correspondant aux led_*_o) sera donc la sortie « < » du comparateur que l'on va inverser.

- D'après la description, le compteur 10 bits n'a pas besoin d'enable ni d'entrées en plus du reset et de la clock. C'est donc un compteur qui va tourner en boucle. Pas besoin de comparateur puisqu'on va de la valeur minimale à la maximale représentable. Le compteur sera donc très simple. On aura un registre 10 bits qui aura comme entrée sa valeur (remise à 0 par le reset). Ce compteur aura deux sorties nécessaires qui sont la fin de la période et la valeur du compteur qui sera branché sur la sortie « cycle_pwn_o ». La fin de la période sera gérée par une comparaison de la valeur du compteur avec la valeur maximale (0x3ff) dont on prend l'égalité. Enfin la valeur du compteur utilisée dans l'explication précédente n'a rien de spécial.

Remarque : Pour être plus propre lors de la transition avec les leds, il faudrait des registres qui ne sont modifiés qu'en fin de période mais cela est plus coûteux car nécessiterait 3 registres. Une période étant vraiment courte, on aurait très peu de temps où la valeur est fausse et l'œil humain ne le remarquera jamais. Ce n'est pas une bonne idée dans notre cas puisque le but est d'être le plus optimisé sur le matériel utilisé.

1.2 Schéma blocs des générateurs PWM

Voici le schéma bloc de la solution que nous avons mis sur papier :

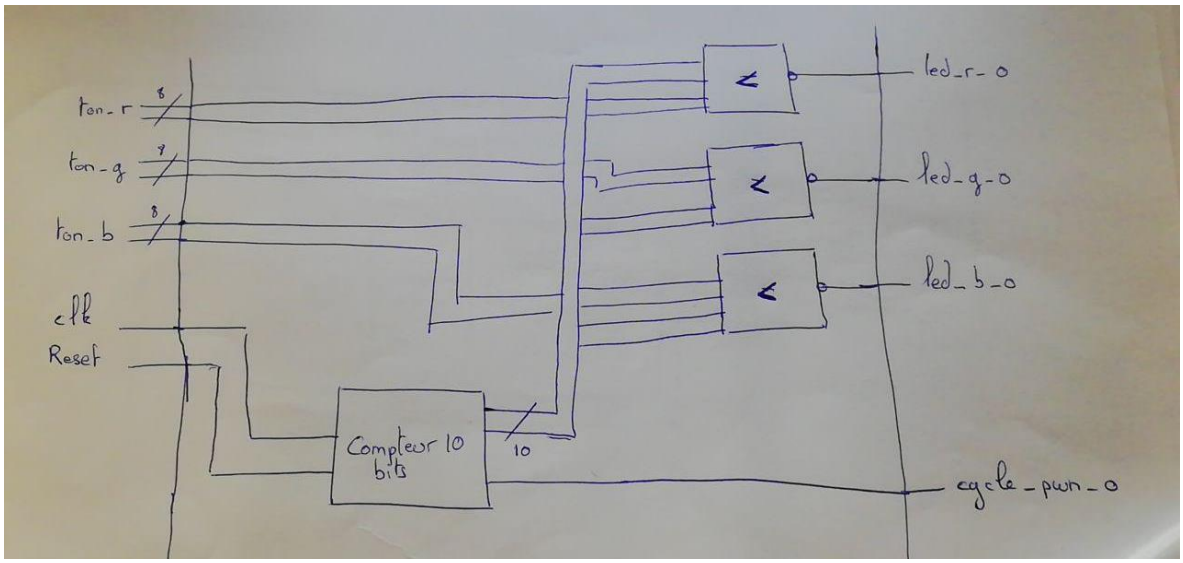


Schéma bloc des trois générateurs PWM

Le schéma est assez clair, on retrouve le compteur, les comparateurs et les entrées et sorties décrites précédemment.
****Note** : Plus tard après l'analyse, nous nous sommes rendu compte qu'il fallait multiplier la valeur des ton_* pour couvrir le « spectre » des 10 bits du compteur avec 8 bits. Ainsi pour faire plus simple, il faudrait faire 3 comparateurs 8 bits et avoir en entrée les 8 bits des ton_* et les 8 bits de poids fort de la valeur du compteur (enlever 2 bits de poids faible du compteur qui entrent dans le comparateur)

Ci-dessous on peut voir le bloc compteur 10 bits présent dans le schéma au-dessus :

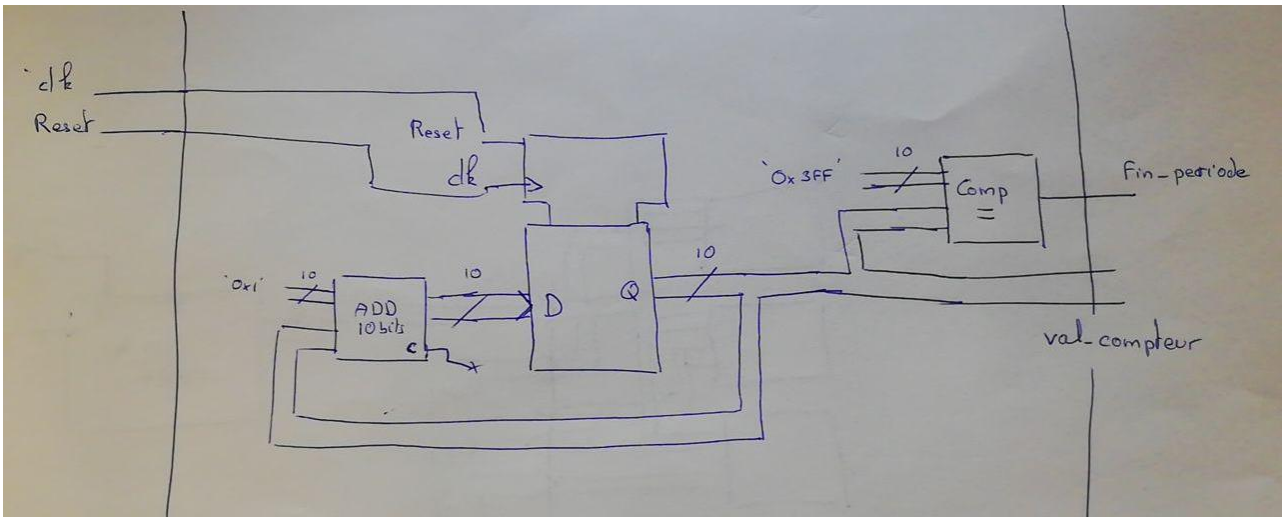


Schéma du compteur 10 bits

1.3 Table des fonctions synchrones du compteur

Table des fonctions synchrones du bloc compteur 10 bits :

Reg_present	Reg_futur
= 0x3ff	0x000
Autres cas	Reg_present + 1

La table est très simple, il n'y a aucune entrée mis à part la clock et le reset asynchrone. C'est un compteur sur lequel on a très peu de contrôle mais qui fait ce qu'on lui demande (et il le fait bien !) pour le circuit mis en place.

1.4 Description VHDL des composants

Note** : Tous les codes sont commentés précisément et les commentaires expliquent précisément l'implémentation et l'utilité de ce que nous avons écrit.

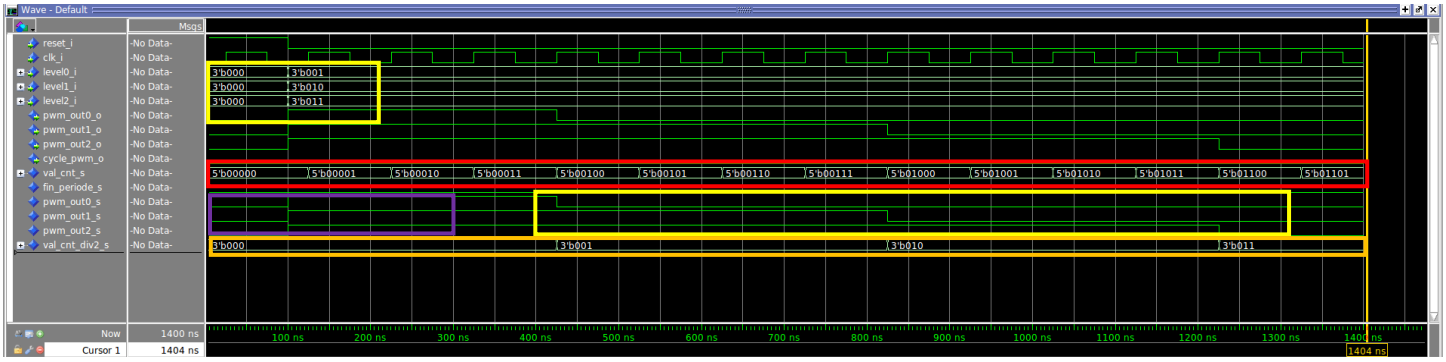
Le premier fichier VHDL que nous avons créée est celui du compteur générique (compteur_N). Nous avons commencé par faire un compteur 10 bits non générique pour tester le fonctionnement. Une fois ce compteur mis en place et fonctionnel, très peu de manipulations étaient nécessaires pour le modifier et en faire un compteur générique.

Pour voir le code du compteur, se référer au fichier « compteur_N.vhd » dans les annexes du rapport.

Pour la description de ce compteur, nous avons écrit ce que nous avions proposé pour le schéma dessiné plus haut.

C'est le seul bloc que nous avons à mettre dans le bloc generateur_3pwm. De fait, dans le fichier vhdl, c'est le seul composant que nous avons utilisé. Pour plus de précisions sur la description, aller voir dans les annexes le fichier « generateur_3pwm.vhd ».

1.5 Vérification du fonctionnement des générateurs PWM



Cadre rouge : On peut vérifier le bon fonctionnement du compteur. Chaque cycle de clock, la valeur est bien incrémentée de 1 jusqu'à la valeur maximale, puis retourne à 0 (on ne le voit pas sur cette image mais cela a bien été testé en initiant la valeur du timer à max valeur – 3).

Cadre violet : On remarque le bon fonctionnement des comparaisons de valeurs. Tant que la valeur du compteur est égale (ou supérieure) la valeur du pwm correspondant est à 0. Les levels sont ensuite initialisées à 001, 010 et 011 avant que le compteur ne modifie sa valeur (restée à 0). Ainsi, toutes les valeurs des levels sont supérieures à celle du compteur et des signaux pwm_out sont tous à 1.

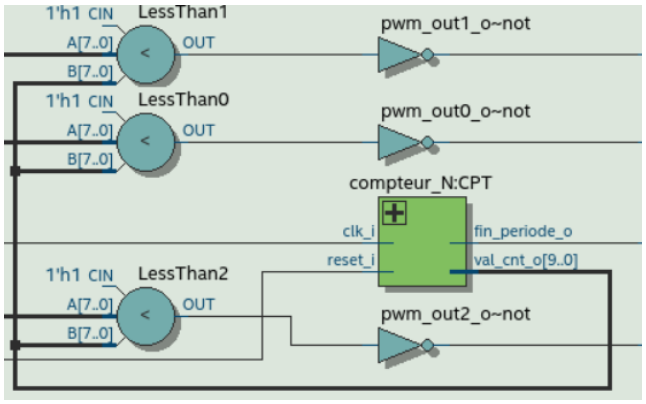
Cadre orange : Nous avons créée un signal supplémentaire dans notre description VHDL pour mieux voir les changements attendus. Pour que les valeurs sur 8 bits couvrent toutes les valeurs sur 10 bits, nous avons la comparaison qui se fait entre la valeur sur 8 bits et les 8 bits de poids fort du compteur. Les changements de ce signal ajouté correspondent en quelque sorte au rythme de comptage pour les valeurs sur 8 bits.

Cadres jaunes : On détecte bien le fonctionnement attendu. Les valeurs sont 001, 010 et 011, on veut donc que les valeurs des pwm_o passent à 0 respectivement lorsque la valeur du compteur passe à 001, 010 et 011, ce qui est le cas. Le temps d'activité des signaux pwm est bien multiplié par quatre (du fait de la comparaison qui met de côté les deux bits de poids faible pour la valeur du compteur). On voit bien 4/8/12 valeurs de compteur avant que les valeurs ne passent à 0.

1.6 Vérification de la description VHDL

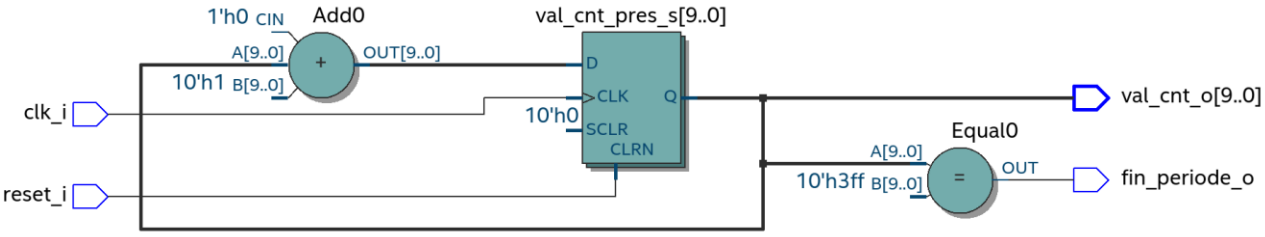
Pour vérifier si notre description VHDL est fidèle à notre analyse, nous avons comparé les vues RTL avec nos schémas.

Pour les générateurs PWM, nous sommes très fidèles et nous retrouvons bien les trois comparateurs 8 bits avec la sortie inversée (NOT) qui donne les bonnes valeurs pour les sorties PWM. Nous voyons aussi le bloc compteur_N que nous avons dans notre schéma. Notre description est donc fidèle à l'analyse que nous avons faite au préalable.



Vue RTL des générateurs PWM

Nous constatons sensiblement la même chose avec le compteur générique. On voit bien les trois éléments : un registre (avec 10 flips flops car 10 bits), un additionneur (pour incrémenter la valeur 1 à 1, addition de la valeur actuelle avec 1) et un comparateur (pour activer un signal qui détermine la fin de la période du compteur, ici 0x3ff).



Vue RTL du compteur 10 bits

Ces vues RTL nous permettent de valider cette partie et de passer à la suite avec une base solide et fonctionnelle.

Pour ce qui est de la quantité de logique, nous avons obtenu les informations suivantes concernant le compteur 10 bits, afin de pouvoir contrôler le nombre de flip flop obtenus

	Resource	Usage
1	▼ Total logic elements	13
1	-- Combinational with no register	3
2	-- Register only	0
3	-- Combinational with a register	10
2		
3	▼ Logic element usage by number of LUT inputs	
1	-- 4 input functions	3
2	-- 3 input functions	0
3	-- 2 input functions	9
4	-- 1 input functions	1
5	-- 0 input functions	0
4		
5	▼ Logic elements by mode	
1	-- normal mode	5
2	-- arithmetic mode	8
3	-- qfbk mode	0
4	-- register cascade mode	0
5	-- synchronous clear/load mode	0
6	-- asynchronous clear/load mode	10
6		
7	Total registers	10
8	Total logic cells in carry chains	9
9	I/O pins	13
10	Maximum fan-out node	clk_i

On peut voir le nombre de flip flop sur cette image dans les cadres rouges.

Il est bien de 10, ce qui correspond au nombre de bits de notre compteur dans ce cas-là. Le nombre de flip-flops correspond donc à notre description.

Il est intéressant de voir que 13 « logic elements » au total sont présent dans ce bloc. Cela inclut nos 10 flip-flops, mais laisse encore trois éléments logiques sans registres.

Si l'additionneur et le comparateur compteur comme 2 éléments logique, notre théorie est que le compteur en lui-même est considérer aussi comme un « logic element ». Cela permet d'expliquer ce chiffre de 3 éléments logiques.

1.7 Tests effectués pour les générateurs PWM

- Test que le compteur revienne bien à 0 et continue de s'incrémenter.

- Tests avec différents sets de valeurs :

    . 000, 010, 100

    . 110, 101, 111

    . 010, 011, 110

    . 111, 111, 111

    . 000, 000, 000

    Tout est fonctionnel et le comportement attendu est le bon.

- Il est important de bien faire un reset asynchrone au début de l'utilisation du système pour que le compteur ait bien des valeurs initialisées et non 'U'.

Ayant testé les valeurs limites, nous ne voyons pas d'autres tests pertinents à effectuer pour la partie des générateurs PWM.

Les tests effectués pour cette partie sont bons, la description est donc validée, tout comme le fonctionnement.

2. Réalisation du module de conversion

2.1 Analyse du module de conversion

Il y a deux parties pour gérer le fonctionnement attendu.

On souhaite premièrement avoir une gestion permettant de choisir entre deux modes : le mode « blanc » et le mode « normal ». Le mode blanc permet d'avoir toutes les variations de blanc que propose le système mis en place sans avoir à modifier toutes les leds (car le blanc est obtenu en ayant la même intensité de rouge, vert et bleu) pour avoir les mêmes valeurs. Ainsi, un signal spécifique va permettre, en ne modifiant qu'une seule valeur d'intensité, d'avoir toutes les variations de blanc. Dans notre système, on va prendre la valeur de l'intensité du vert pour gérer le mode blanc.
Le but est donc que si le signal m_blanc est actif, les valeurs d'intensité du rouge et du bleu soient les mêmes que la valeur d'intensité du vert. On utilisera donc m_blanc pour sélectionner la valeur du vert si on souhaite le mode blanc et la valeur des couleurs respectives si ce n'est pas le cas.

L'autre partie consiste à convertir la valeur d'intensité donnée sur 3 bits en une valeur de ton qui elle est sur 8 bits. On utilise donc le tableau suivant pour la conversion :

Niveau intensité, 3bits	Ton (level PWM), 8 bits	
0	0	0x00
1	5	0x05
2	21	0x15
3	37	0x25
4	53	0x35
5	69	0x45
6	85	0x55
7	101	0x65

On remarque que les valeurs des tons que l'on prend peuvent être codées sur 7 bits. Il n'est pas nécessaire d'étendre plus le spectre des valeurs car au-dessus de ces valeurs, on ne voit plus vraiment la différence à l'œil nu. Déjà à 0x65 c'est trop intense.

L'idée est d'avoir le niveau d'intensité qui est le sélecteur des 8 valeurs possibles sur 8 bits des ton_*.

Rien de compliqué pour cette partie en réalité.

2.2 Schéma bloc du module de conversion

Après l'analyse de la gestion du mode blanc, voici la solution que nous avons pensé. C'est une partie assez simple qui ne nécessite pas vraiment de schéma bloc mais nous avons tout de même réalisé ce schéma pour le comparer à la vue RTL une fois la description VHDL synthétisée.

On voit ci-dessous en entrée du MUX les niveaux d'intensité des trois couleurs (rouge vert et bleu). Le vert est de toute façon la bonne valeur, mais pour les deux autres couleurs, c'est m_blanc qui pilote le MUX qui prend comme entrée la couleur ou la valeur d'intensité du vert suivant le mode choisi au moment T.

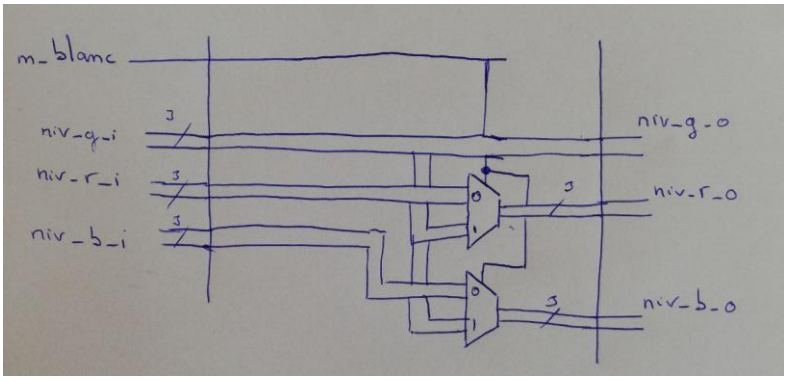


Schéma de la gestion du mode blanc

Ensuite les valeurs de niveau d'intensité sont utilisées comme bits de sélection d'un mux 3 à 8. Les valeurs de sélection sont les constantes définies pour représenter le spectre des valeurs. Comme les valeurs choisies sont représentables sur 7 bits, on concatène un 0 en bit de poids fort à la sortie du MUX pour fournir une valeur sur 8 bits.

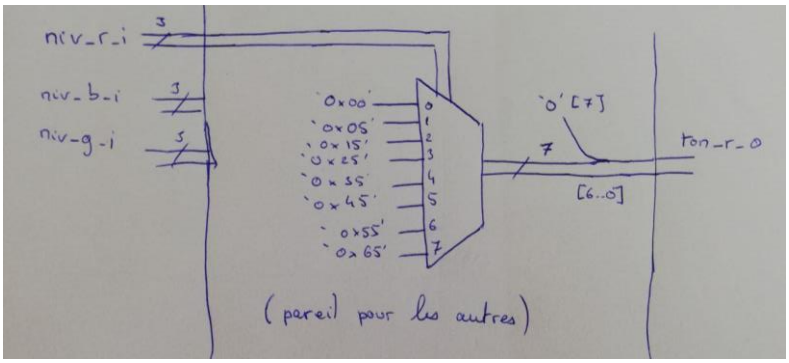


Schéma du module de conversion des intensités

Ci-dessus on ne voit qu'un des MUX mais dans la réalité il faut 3 MUX et chacun de ces MUX est piloté par la valeur d'intensité du rouge, bleu et vert. Les constantes sont les mêmes pour les trois valeurs de ton_* que l'on fournira en sortie.

2.3 Description VHDL du module de conversion

Pour cette partie nous avons deux blocs différents pour le traitement d'un côté du mode blanc et un autre pour la conversion de l'intensité sur 3 bits en valeur 8 bit d'après la description décrite précédemment.

Ainsi nous avons un fichier « mode_blanc.vhd » consultable en annexe qui décrit le bloc de gestion du mode. Nous n'avons pas grand-chose à dire sur ce fichier, il est très simple et ne nécessite pas beaucoup d'explications.

Nous avons aussi la description, consultable dans les annexes, de la conversion des niveaux. Le fichier est « level_converter.vhd ». De même que le fichier précédent c'est concrètement la description de 3 MUX pareils.

2.4 Vérification du fonctionnement du module de conversion

```
# vsim -voptargs=""+acc"" work.LED_RGB_tb
# Start time: 10:25:06 on Nov 25,2019
# ** Note: (vsim-8009) Loading existing optimized design _opt1
# Loading std.standard
# Loading std.textio(body)
# Loading ieee.std_logic_1164(body)
# Loading ieee.numeric_std(body)
# Loading work.led_rgb_tb(test_bench)#1
# Loading work.led_rgb_top(struct)#1
# Loading work.gestion_mode(flot_don)#1
# Loading work.level_converter(flot_don)#1
# Loading work.gen_3pwm(struct)#1
# Loading work.compteur_n(comport)#1
VSIM 2> run -all
# ** Note: Debut de la simulation
#    Time: 0 ns  Iteration: 0  Instance: /led_rgb_tb
# ** Warning: NUMERIC_STD."<=": metavalue detected, returning FALSE
#    Time: 0 ns  Iteration: 0  Instance: /led_rgb_tb/uut/Bloc_pwm
# ** Warning: NUMERIC_STD."<=": metavalue detected, returning FALSE
#    Time: 0 ns  Iteration: 0  Instance: /led_rgb_tb/uut/Bloc_pwm
# ** Warning: NUMERIC_STD."<=": metavalue detected, returning FALSE
#    Time: 0 ns  Iteration: 0  Instance: /led_rgb_tb/uut/Bloc_pwm
# ** Warning: NUMERIC_STD."=": metavalue detected, returning FALSE
#    Time: 0 ns  Iteration: 0  Instance: /led_rgb_tb/uut/Bloc_pwm/CPT
# ** Warning: NUMERIC_STD."<=": metavalue detected, returning FALSE
#    Time: 0 ns  Iteration: 1  Instance: /led_rgb_tb/uut/Bloc_pwm
# ** Warning: NUMERIC_STD."<=": metavalue detected, returning FALSE
#    Time: 0 ns  Iteration: 1  Instance: /led_rgb_tb/uut/Bloc_pwm
# ** Warning: NUMERIC_STD."<=": metavalue detected, returning FALSE
#    Time: 0 ns  Iteration: 1  Instance: /led_rgb_tb/uut/Bloc_pwm
# ** Warning: NUMERIC_STD."<=": metavalue detected, returning FALSE
#    Time: 0 ns  Iteration: 2  Instance: /led_rgb_tb/uut/Bloc_pwm
# ** Warning: NUMERIC_STD."<=": metavalue detected, returning FALSE
#    Time: 0 ns  Iteration: 2  Instance: /led_rgb_tb/uut/Bloc_pwm
# ** Warning: NUMERIC_STD."<=": metavalue detected, returning FALSE
#    Time: 0 ns  Iteration: 2  Instance: /led_rgb_tb/uut/Bloc_pwm
# ** Warning: NUMERIC_STD."<=": metavalue detected, returning FALSE
#    Time: 0 ns  Iteration: 4  Instance: /led_rgb_tb/uut/Bloc_pwm
# ** Note: >>Nombre d'erreur détecté = 0
#    Time: 801337 ns  Iteration: 0  Instance: /led_rgb_tb
# ** Note: >>Fin de la simulation
#    Time: 801337 ns  Iteration: 0  Instance: /led_rgb_tb

VSIM 3>
```

Un test bench nous était fourni pour cette partie, nous l'avons donc lancé et aucune erreur n'a été relevée. Le fonctionnement que nous proposons et donc celui qui est attendu de la part du mandataire (vous !).

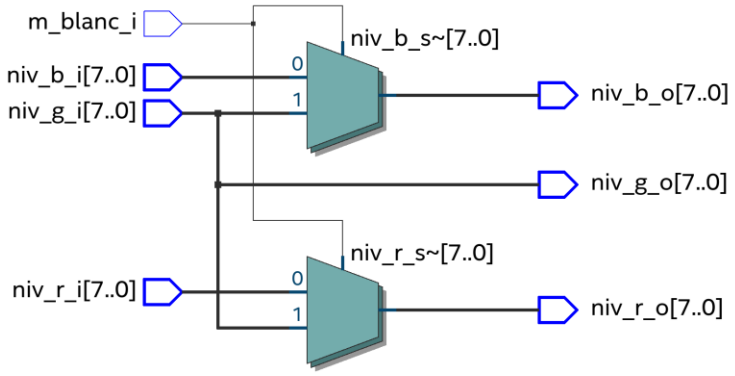
Cette partie étant très simple, nous n'avons pas effectué de tests supplémentaires nous-même car le test bench couvrait déjà toutes les possibilités de fonctionnement.

La description VHDL de cette partie est donc validée, tout comme son interaction avec la partie précédente.

2.5 Vérification de la description VHDL

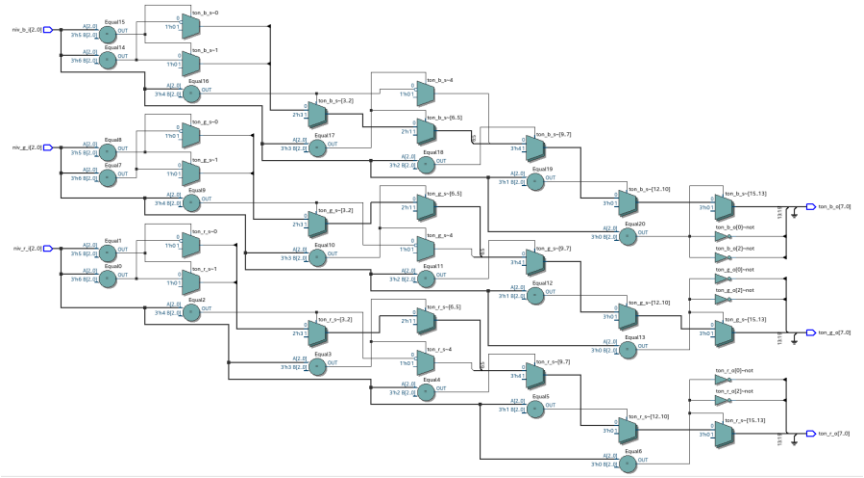
Pour les mêmes raisons que précédemment, nous allons consulter les vues RTL pour vérifier si notre description VHDL est bien similaire à notre analyse.

Sur la vue ci-dessous, nous voyons bien la ressemblance avec notre schéma papier. Deux MUX pour le choix des couleurs bleu et rouge qui sont déterminé par le signal m_blanc_i.



Vue RTL du mode blanc

Sur la vue suivante, on note une légère différence qui est présente car nous avons utilisé dans notre schéma un MUX 3 à 8 tandis que sur la vue RTL ne sont utilisé que des MUX 2 à 1. Mais en analysant un peu ce que fait chaque partie, on voit que c'est un fonctionnement équivalent à notre analyse.



Vue RTL du module de conversion

En analysant la quantité de logique obtenue, on peut observer les choses suivantes :

	Resource	Usage
1	▼ Total logic elements	12
1	-- Combinational with no register	12
2	-- Register only	0
3	-- Combinational with a register	0
2		
3	▼ Logic element usage by number of LUT inputs	
1	-- 4 input functions	0
2	-- 3 input functions	12
3	-- 2 input functions	0
4	-- 1 input functions	0
5	-- 0 input functions	0
4		
5	▼ Logic elements by mode	
1	-- normal mode	12
2	-- arithmetic mode	0
3	-- qfbk mode	0
4	-- register cascade mode	0
5	-- synchronous clear/load mode	0
6	-- asynchronous clear/load mode	0
6		
7	Total registers	0
8	I/O pins	33
9	Maximum fan-out node	niv_r_i[2]
10	Maximum fan-out	4

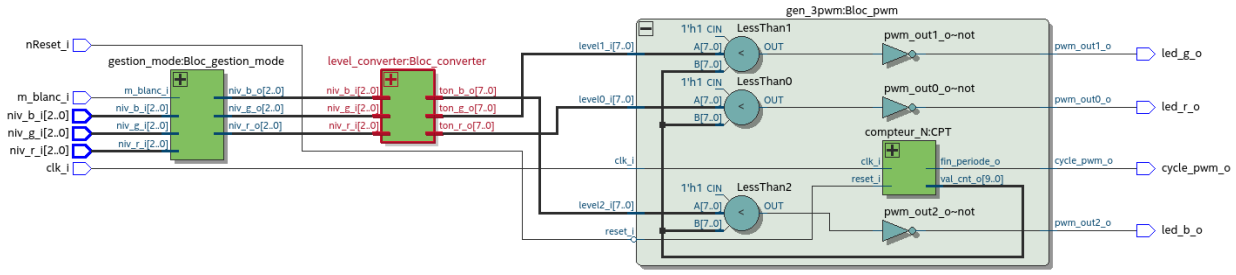
On commence par voir que 12 éléments logiques sont utilisés.

En observant un peu plus, on s'aperçoit que ces éléments logiques sont des fonctions avec trois entrées.

Cela vient du fait que chaque entrée niveau utilise 4 de ces éléments logiques afin de réussir à transformer le niveau.

Même si le fonctionnement peut paraitre cryptique sur la vue RTL du module de conversion, l'essentiel est là : Le synthétiseur à optimiser la quantité de logique utilisée par rapport à notre description.

La dernière partie était principalement une application pratique d'intégration sur la carte de notre système. Si tout s'est passé sans problème pour ce qui est de l'intégration, nous devions quand même discuter de la quantité de logique obtenue pour le système en entier. Vous trouverez ci-dessous la vue RTL de « led_rgb_top.vhd », qui n'était pas demandée mais qui aide à mieux visualiser le système, et juste en dessous la quantité de logique avec l'analyse l'accompagnant.



Vue RTL du système avec gen_3pwm étendu

	Resource	Usage
1	▼ Total logic elements	38
1	-- Combinational with no register	28
2	-- Register only	0
3	-- Combinational with a register	10
2		
3	▼ Logic element usage by number of LUT inputs	
1	-- 4 input functions	20
2	-- 3 input functions	8
3	-- 2 input functions	9
4	-- 1 input functions	1
5	-- 0 input functions	0
4		
5	▼ Logic elements by mode	
1	-- normal mode	30
2	-- arithmetic mode	8
3	-- qfbk mode	0
4	-- register cascade mode	0
5	-- synchronous clear/load mode	0
6	-- asynchronous clear/load mode	10
6		
7	Total registers	10
8	Total logic cells in carry chains	9
9	I/O pins	16
10	Maximum fan-out node	gen_2...s[6]

Nous avons remarqué deux choses qu'il est intéressant de relever :

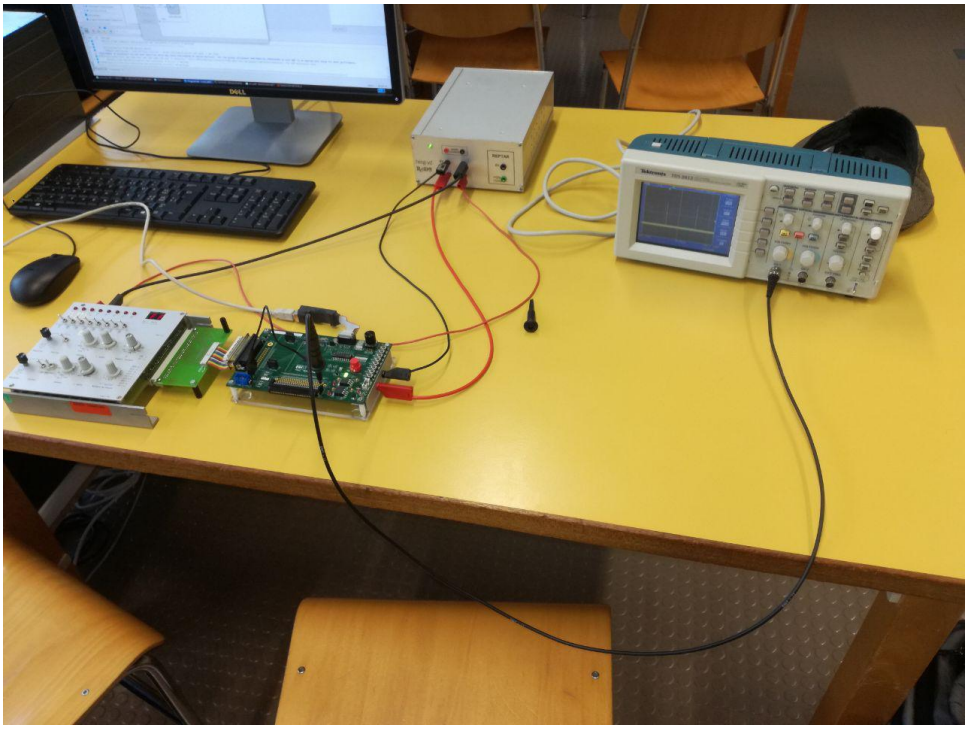
Le nombre de flip-flops est toujours le même, ce qui est un signe de bonne santé du système, la seule partie devant utiliser des flip-flops étant le compteur N bits, cela nous prouve qu'il n'y en a pas d'autres utilisés.

En observant le nombre de fonctions avec 3 entrées, on voit qu'il y en a 8, au lieu des 12 qui étaient nécessaires au module de conversion.

Cela peut venir d'optimisation, mais cela nous montre surtout que le nombre d'éléments logique du top n'est pas juste une addition sans optimisation des logiques de chaque bloc.

2.6 Tests effectués sur le système final

Tout d'abord, voici une image de l'installation que nous avons utilisé pour effectuer les tests de notre implémentation :



Pour commencer, nous avons utilisé l'oscilloscope pour vérifier que la fréquence était bien la bonne. Avec le calcul 1/frequence = période, on retrouve bien les 1024 coups de clock pour une période :

TESTS DU FONCTIONNEMENT :

- Test des différentes couleurs :

    . Affichage du bleu (et des 8 intensités différentes)

    . Affichage du rouge (et des 8 intensités différentes)

    . Affichage du vert (et des 8 intensités différentes)

    . Affichage du cyan (avec différentes intensités)

    . Affichage du magenta (avec différentes intensités)

    . Affichage du jaune (avec différentes intensités)

    . Affichage du blanc (et des 8 intensités différentes)

Images de certaines couleurs (vert, rouge, blanc, bleu, magenta) :





- Si toutes les valeurs de verts sont à 0 mais que les valeurs des deux autres couleurs sont différentes de 0, on obtient du magenta en mode normal et la led est éteinte lorsque le bouton du mode blanc est pressé.

- Les très hautes valeurs d'intensités sont plus difficiles à différencier à l'œil nu mais sont tout de même différentes.

Conclusion

1. Difficultés rencontrées

- Compréhension des objectifs du laboratoire (Gestion de la fréquence). En effet, la compréhension du laboratoire nous a pris beaucoup de temps au début, n'ayant jamais vu auparavant le fonctionnement d'un PWM.
- Gestion de la généricité pour le compteur et le bloc de gestion PWM. La mise en place du programme générique depuis des blocs prévus à la base pour prendre un nombre de bits en entrée statique nous à donné du fil à retordre.

2. Compétences acquises

- Décomposition du système en bloc VHDL différents
- Consolidation de la méthodologie de test, compilation, synthèse et intégration.

3. Résultat obtenu

Nous avons réussi à mettre en place toutes les étapes qui nous étaient demandé dans ce laboratoire. Tous les fichiers vhdl sont fonctionnels et l'expérience a été présenter à Mr. Masle. Les objectifs sont donc atteints.

Annexes

compteur_N.vhd

```
--------------------------------------------------------------------------------
-- HEIG-VD, Haute Ecole d'Ingenierie et de Gestion du canton de Vaud
-- Institut REDS, Reconfigurable & Embedded Digital Systems
--
-- Fichier      : compteur_N.vhd
--
-- Description  : Bloc représentant un compteur N bits
--
-- Auteur       : Pierrick Muller
-- Date         : 21.11.2019
-- Version      : 0.1
--
-- Utilise      : gen_3pwm.vhd
--
--| Modifications |-------------------------------------------------------------
-- Version   Auteur Date               Description
-- 0.1       PMR    21.11.2019         Création de la logique vdhl
--------------------------------------------------------------------------------

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

entity compteur_N is
   generic(N_CNT : positive range 1 to 16 := 10);
   port(reset_i     : in  std_logic; -- entrée reset asynchrone
        clk_i       : in  std_logic; -- entrée clock
        val_cnt_o   : out  std_logic_vector(N_CNT-1 downto 0); -- sortie valeur
du compteur
        fin_periode_o  : out std_logic -- Sortie fin d'une période de compteur
(Atteint valeur max)
       );
end compteur_N;

architecture comport of compteur_N is

  -- components declaration
  -- Pas de composants externes

 -- declaration internal signals
 -- Comme on souhaite créer un compteur générique, la taille des vecteurs
signaux
 -- correspondant à l'état présent et à l'état futur correspond à la taille
 -- de la sortie val_cnt_o qui est détérminée en fonction du générique N_CNT
 signal val_cnt_pres_s : std_logic_vector((val_cnt_o'length - 1) downto 0);
 signal val_cnt_fut_s : std_logic_vector((val_cnt_o'length - 1) downto 0);
begin

-- Décodeur d'état futur
```

```
-- On se contente d'ajouter 1 à la valeur actuelle du compteur, ce bloc sera
-- transformé en additionneur, comme on peut le voir dans la vue RTL
val_cnt_fut_s <= std_logic_vector(unsigned(val_cnt_pres_s) + 1);

-- Ce process correspond à notre registre. La gestion du reset asynchrone est
-- directement inclut dedans, pour ce qui est du reste on s'occupe juste de
-- stocker la valeur du compteur futur à la place de la valeur du compteur
-- présent à chaque tick d'horloge
mem: process (clk_i,reset_i)
begin
  if(reset_i = '1') then
    val_cnt_pres_s <= (others => '0');
  elsif rising_Edge(clk_i) then
    val_cnt_pres_s <= val_cnt_fut_s;
  end if;
end process;

-- Décodeur de sortie
-- On gère la sortie détérminant la fin de la periode et donc la fréquence de
-- notre compteur en définissant la sortie avec la valeur 1 si la valeur du
-- compteur est égal à la valeur max que le compteur peut prendre
  fin_periode_o <= '1' when unsigned(val_cnt_pres_s) = (2**(val_cnt_o'length)) -
1 else   --(2**N)-1
                   '0';

  -- On assigne la valeur du signal représentant la valeur du compteur présent
  -- à la sortie de notre bloc compteur.
  val_cnt_o <= val_cnt_pres_s;



end comport;
```

gen_3pwm.vhd

```
--------------------------------------------------------------------------------
-- HEIG-VD, Haute Ecole d'Ingenierie et de Gestion du canton de Vaud
-- Institut REDS, Reconfigurable & Embedded Digital Systems
--
-- Fichier      : gen_3pwm.vhd
--
-- Description  : Bloc comprenant 3 generateurs PWM
--                clk_sys = 1 MHz, Fpwm =976 Hz, soit periode 1.024 ms
--
-- Auteur       : Etienne Messerli
-- Date         : 10.04.2017
-- Version      : 0.0
--
-- Utilise      : Manipulation Commande d'une led RGB, cours CSN
--
--| Modifications |------------------------------------------------------------
-- Version   Auteur Date               Description
-- 0.1       PMR    14.11.2019         Ajout de la logique vhdl
--------------------------------------------------------------------------------

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

```vhdl
use ieee.numeric_std.all;

entity gen_3pwm is
   generic(N_PWM : positive range 1 to 16 := 8);
   port(reset_i     : in  std_logic; -- entrée reset asynchrone
        clk_i       : in  std_logic; -- entrée clock
        -- vecteurs représentant les différents levels des couleurs (R,G,B),
        -- de taille N_PWM
        level0_i    : in  std_logic_vector(N_PWM-1 downto 0);
        level1_i    : in  std_logic_vector(N_PWM-1 downto 0);
        level2_i    : in  std_logic_vector(N_PWM-1 downto 0);
        -- Sortie représentant le level en fonction de la durée durant laquelle
        -- ils sont activés (principe pwm)
        pwm_out0_o  : out std_logic;
        pwm_out1_o  : out std_logic;
        pwm_out2_o  : out std_logic;
        -- Sortie représentant la fin d'un cycle de compteur
        -- (Utilisé pour la fréquence)
        cycle_pwm_o : out std_logic
       );
end gen_3pwm;

architecture struct of gen_3pwm is

  --components declaration
  -- Déclaration d'un compteur N bits
  component compteur_N
  generic(N_CNT : Positive range 1 to 16);
  port (reset_i     : in  std_logic; -- entrée reset asynchrone
        clk_i       : in  std_logic; -- entrée clock
        val_cnt_o   : out  std_logic_vector(N_CNT-1 downto 0); -- sortie valeur
du compteur
        fin_periode_o  : out std_logic -- Sortie fin d'une période de compteur
(Atteint valeur max)
        );

  end component;

  for all : compteur_N use entity work.compteur_N(comport);

 --declaration internal signals
  -- Comme on ne peut pas utilisé la valeur N_CNT du compteur pour savoir
  -- combien de bits sont nécéssaires pour stocker la valeur du compteur,
  -- on se base sur N_PWM pour définir la taille de ce vecteur. En effet,
  -- en restant sur une utilisation simple, on part du principe que le
  -- rapport entre la taille du compteur et la taille des levels en entrée
  -- sera toujours de deux, afin de garantir un bon fonctionnement du programme.
  signal val_cnt_s :  std_logic_vector((N_PWM + 2) - 1 downto 0);
  -- Signaux stockant les valeurs internes et les valeurs qui seront assignée
  -- aux sorties de notre block gen_3pwm.
  signal fin_periode_s     : std_logic;
  signal pwm_out0_s  :  std_logic;
  signal pwm_out1_s  :  std_logic;
  signal pwm_out2_s  :  std_logic;
  -- Signal permettant de stocker la valeur du compteur divisé par deux afin d
  -- pouvoir la comparer aux valeurs des levels tout en profitant de tous le
  -- spectre de valeurs de notre compteur.
  signal val_cnt_div2_s  : std_logic_vector((N_PWM + 2) - 3 downto 0);
```

```
begin

 CPT : compteur_N
 -- Comme expliqué dans le commentaire au dessus de val_cnt_s, la taille en
 -- nombre de bits du compteur est égal à la taille des levels en entrée + 2
 -- (dans notre utilisation, cette taille de compteur est de 10)
 generic map (N_CNT => N_PWM + 2)
 port map (
       -- assignation des I/O du compteur aux différents signaux
       reset_i  => reset_i,
       clk_i  => clk_i,
       val_cnt_o => val_cnt_s,
       fin_periode_o => fin_periode_s
 );

 -- On divise par deux la valeur du compteur afin de faire les comparaisons
 -- avec les valeurs des levels par la suite
 val_cnt_div2_s <= val_cnt_s((val_cnt_s'length - 1) downto 2);
 -- Comme dans le schéma, on effectue une comparaison par level avec la valeur
 -- du compteur. On assigne la valeur 0 au signal si la valeur contenue dans
 -- le level est plus petite ou egale à la valeur du compteur, sinon, le signal
 -- pwm correspondant doit rester à 1.
 pwm_out0_s <=  '0' when unsigned(level0_i) <= unsigned(val_cnt_div2_s) else
                '1';
 pwm_out1_s <=  '0' when unsigned(level1_i) <= unsigned(val_cnt_div2_s) else
                '1';
 pwm_out2_s <=  '0' when unsigned(level2_i) <= unsigned(val_cnt_div2_s) else
                '1';

 -- On assigne les signaux aux differentes sorties de notre bloc
 cycle_pwm_o <= fin_periode_s;
 pwm_out0_o <= pwm_out0_s;
 pwm_out1_o <= pwm_out1_s;
 pwm_out2_o <= pwm_out2_s;


end struct;
```

gestion_mode.vhd

```
--------------------------------------------------------------------------------
-- HEIG-VD, Haute Ecole d'Ingenierie et de Gestion du canton de Vaud
-- Institut REDS, Reconfigurable & Embedded Digital Systems
--
-- Fichier      : gestion_mode.vhd
--
-- Description  : Gestion des niveaux d'intensite avec mode blanc
--
-- Auteur       : Etienne Messerli
-- Date         : 14.11.2019
-- Version      : 1.0
--
-- Utilise      :
--
--| Modifications |-------------------------------------------------------------
```

```
-- Version   Auteur Date               Description
--
--------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity gestion_mode is
   generic(N_LEVEL : positive range 1 to 16:= 8);  --nombre de bits PWM
   port(m_blanc_i : in  std_logic; -- Entrée représentant le mode (1 = mode
blanc, 0 = normal)
        -- Entrée représentant les niveau de bleu, vert et gris
        niv_r_i   : in  std_logic_vector(N_LEVEL-1 downto 0);
        niv_g_i   : in  std_logic_vector(N_LEVEL-1 downto 0);
        niv_b_i   : in  std_logic_vector(N_LEVEL-1 downto 0);
        -- Sortie représentant les niveau de bleu, vert et gris après gestion du
mode
        niv_r_o   : out std_logic_vector(N_LEVEL-1 downto 0);
        niv_g_o   : out std_logic_vector(N_LEVEL-1 downto 0);
        niv_b_o   : out std_logic_vector(N_LEVEL-1 downto 0)
       );
end gestion_mode;

architecture flot_don of gestion_mode is

  -- Signaux permettant par la suite d'affecter les sorties.
  signal niv_r_s : std_logic_vector(N_LEVEL-1 downto 0);
  signal niv_b_s : std_logic_vector(N_LEVEL-1 downto 0);

begin
  -- Représentation des deux muxs permettant de définir le mode, comme démontré
  -- dans le schéma bloc fournit dans le rapport.
  niv_r_s <= niv_g_i when m_blanc_i = '1' else
         niv_r_i;
  niv_b_s <= niv_g_i when m_blanc_i = '1' else
         niv_b_i;

  -- Assignation des sorties.
  niv_r_o <= niv_r_s;
  niv_b_o <= niv_b_s;
  niv_g_o <= niv_g_i;

end flot_don;
```

level_converter.vhd

```
--------------------------------------------------------------------------------
-- HEIG-VD, Haute Ecole d'Ingenierie et de Gestion du canton de Vaud
-- Institut REDS, Reconfigurable & Embedded Digital Systems
--
-- Fichier      : level_converter.vhd
--
-- Description  : Conversion d'un niveau d'une LED en temps on (T on)
--
-- Auteur       : Gaetan Matthey
-- Date         : 27.03.2017
```

```
-- Version      : 1.0
--
-- Utilise      :
--
--| Modifications |------------------------------------------------------
-- Version   Auteur Date               Description
-- 1.1       EMI    23.09.2019      Supprime generic
-------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity level_converter is
   -- Generique sont supprime car ne compile pas dans Questasim ! EMI 23.09.2019
   --generic(N_LEVEL  : positive range 1 to 16 := 3;
   --        N_PWM : positive range 1 to 16 := 8);
   port(
        -- Entrée représentant les niveaux de bleu, vert et rouge
        niv_r_i   : in  std_logic_vector(2 downto 0);
        niv_g_i   : in  std_logic_vector(2 downto 0);
        niv_b_i   : in  std_logic_vector(2 downto 0);
        -- Sortie après conversion en level de bleu, vert et rouge
        ton_r_o   : out std_logic_vector(7 downto 0);
        ton_g_o   : out std_logic_vector(7 downto 0);
        ton_b_o   : out std_logic_vector(7 downto 0)
       );
end level_converter;

architecture flot_don of level_converter is

  -- Signaux représentant les sorties après conversion en level de bleu, vert et
rouge
  signal ton_r_s : std_logic_vector(6 downto 0);
  signal ton_g_s : std_logic_vector(6 downto 0);
  signal ton_b_s : std_logic_vector(6 downto 0);

begin

  -- Assignation du level de rouge à l'aide de when else
  ton_r_s <= "0000000" when niv_r_i = "000" else
         "0000101" when niv_r_i = "001" else
         "0010101" when niv_r_i = "010" else
         "0100101" when niv_r_i = "011" else
         "0110101" when niv_r_i = "100" else
         "1000101" when niv_r_i = "101" else
         "1010101" when niv_r_i = "110" else
         "1100101" when niv_r_i = "111" else
         "-------";
  -- Assignation du level de vert à l'aide de when else
  ton_g_s <= "0000000" when niv_g_i = "000" else
         "0000101" when niv_g_i = "001" else
         "0010101" when niv_g_i = "010" else
         "0100101" when niv_g_i = "011" else
         "0110101" when niv_g_i = "100" else
         "1000101" when niv_g_i = "101" else
         "1010101" when niv_g_i = "110" else
         "1100101" when niv_g_i = "111" else
         "-------";
```

```
  -- Assignation du level de bleu à l'aide de when else
  ton_b_s <= "0000000" when niv_b_i = "000" else
         "0000101" when niv_b_i = "001" else
         "0010101" when niv_b_i = "010" else
         "0100101" when niv_b_i = "011" else
         "0110101" when niv_b_i = "100" else
         "1000101" when niv_b_i = "101" else
         "1010101" when niv_b_i = "110" else
         "1100101" when niv_b_i = "111" else
         "-------";

  -- Assignation des sorties et ajout d'un 0 au début (optimisation)
  ton_r_o <= '0' & ton_r_s;
  ton_g_o <= '0' & ton_g_s;
  ton_b_o <= '0' & ton_b_s;


end flot_don;
```

led_rgb_top.vhd

```
--------------------------------------------------------------------------------
-- HEIG-VD, Haute Ecole d'Ingenierie et de Gestion du canton de Vaud
-- Institut REDS, Reconfigurable & Embedded Digital Systems
--
-- Fichier      : led_rgb_top.vhd
--
-- Description  : Top du projet de commande Led RGB avec PWM
--
-- Auteur       : Etienne Messerli
-- Date         : 16.11.2015
-- Version      : 0.0
--
-- Utilise      : Manipulation Commande d'une led RGB, cours CSN
--
--| Modifications |------------------------------------------------------------
-- Version   Auteur Date               Description
--  1.0       GAA    23.03.2017      Ajout d'un level converter pour adapter la
luminosité de la LED (luminosité linéaire)
--  1.1       EMI    23.09.2019      Supprime generic composant
"level_converter"
--  1.2       SMS    11.11.2019      supprime composant ctrl_level_rgb, les
valeurs
--                                   des pwm arrivent directement en entrées
--  1.3       EMI    14.11.2019      Ajout composant gestion_mode
--                                   Ajout generique N_LEVEL dans entité
--
--------------------------------------------------------------------------------

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

entity led_rgb_top is
```

```
    generic(N_LEVEL : positive range 1 to 16 := 3;  --nombre de bits niveau
intensité
            N_PWM   : positive range 1 to 16 := 8);    --nombre de bits PWM
    port(nReset_i     : in std_logic;
         clk_i        : in  std_logic;
         m_blanc_i    : in  std_logic;
         niv_r_i      : in  std_logic_vector(N_LEVEL-1 downto 0);
         niv_g_i      : in  std_logic_vector(N_LEVEL-1 downto 0);
         niv_b_i      : in  std_logic_vector(N_LEVEL-1 downto 0);
         led_r_o      : out std_logic;
         led_g_o      : out std_logic;
         led_b_o      : out std_logic;
         cycle_pwm_o  : out std_logic
    );
end led_rgb_top;

architecture struct of led_rgb_top is

  --components declaration

   component gestion_mode
      generic(N_LEVEL : positive range 1 to 16:= 8);  --nombre de bits PWM
      port(m_blanc_i : in  std_logic;
           niv_r_i   : in  std_logic_vector(N_LEVEL-1 downto 0);
           niv_g_i   : in  std_logic_vector(N_LEVEL-1 downto 0);
           niv_b_i   : in  std_logic_vector(N_LEVEL-1 downto 0);
           niv_r_o   : out std_logic_vector(N_LEVEL-1 downto 0);
           niv_g_o   : out std_logic_vector(N_LEVEL-1 downto 0);
           niv_b_o   : out std_logic_vector(N_LEVEL-1 downto 0)
       );
    end component;
    for all: gestion_mode use entity work.gestion_mode(flot_don);


    component gen_3pwm
        generic(N_PWM : positive range 1 to 16:= 8);  --nombre de bits PWM
        port(reset_i     : in  std_logic;
             clk_i       : in  std_logic;
             level0_i    : in  std_logic_vector(N_PWM-1 downto 0);
             level1_i    : in  std_logic_vector(N_PWM-1 downto 0);
             level2_i    : in  std_logic_vector(N_PWM-1 downto 0);
             pwm_out0_o  : out std_logic;
             pwm_out1_o  : out std_logic;
             pwm_out2_o  : out std_logic;
             cycle_pwm_o : out std_logic
        );
    end component;
    for all: gen_3pwm use entity work.gen_3pwm(struct);


    component level_converter is
    -- Generique sont supprime car ne compile pas dans Questasim ! EMI
23.09.2019
    --generic(N_LEVEL  : positive range 1 to 16 := 3;   --nombre de bit niveau
intensite
    --        N_PWM    : positive range 1 to 16 := 8);  --nombre de bits PWM
        port(niv_r_i  : in  std_logic_vector(2 downto 0);
             niv_g_i  : in  std_logic_vector(2 downto 0);
```

```
            niv_b_i  : in  std_logic_vector(2 downto 0);
            ton_r_o  : out std_logic_vector(7 downto 0);
            ton_g_o  : out std_logic_vector(7 downto 0);
            ton_b_o  : out std_logic_vector(7 downto 0)
           );
    end component;
    for all: level_converter use entity work.level_converter(flot_don);

    --internal signals
    signal reset_s : std_logic;
    signal niv_r_s : std_logic_vector(N_LEVEL-1 downto 0);
    signal niv_g_s : std_logic_vector(N_LEVEL-1 downto 0);
    signal niv_b_s : std_logic_vector(N_LEVEL-1 downto 0);
    signal ton_r_s : std_logic_vector(N_PWM-1 downto 0);
    signal ton_g_s : std_logic_vector(N_PWM-1 downto 0);
    signal ton_b_s : std_logic_vector(N_PWM-1 downto 0);


begin

    --adaptation polarite
    reset_s <= not nReset_i;

    Bloc_gestion_mode: gestion_mode
    generic map(N_LEVEL => N_LEVEL)
    port map(m_blanc_i  => m_blanc_i,
             niv_r_i    => niv_r_i,
             niv_g_i    => niv_g_i,
             niv_b_i    => niv_b_i,
             niv_r_o    => niv_r_s,
             niv_g_o    => niv_g_s,
             niv_b_o    => niv_b_s
    );


    Bloc_converter: level_converter
    --generic map(N_LEVEL => 3,
    --           N_PWM   => N_PWM  )
    port map(niv_r_i => niv_r_s,
             niv_g_i => niv_g_s,
             niv_b_i => niv_b_s,
             ton_r_o => ton_r_s,
             ton_g_o => ton_g_s,
             ton_b_o => ton_b_s
    );


    Bloc_pwm: gen_3pwm
    generic map(N_PWM => N_PWM)
    port map(reset_i     => reset_s,
             clk_i       => clk_i,
             level0_i    => ton_r_s,
             level1_i    => ton_g_s,
             level2_i    => ton_b_s,
             pwm_out0_o  => led_r_o,
             pwm_out1_o  => led_g_o,
             pwm_out2_o  => led_b_o,
             cycle_pwm_o => cycle_pwm_o
```

```
    );

end struct;
```