UNIVERSITÉ TOULOUSE III PAUL SABATIER
Université de Toulouse

TP Synthèse et mise en œuvre de systèmes: Pilote automatique de barre franche

Amadou ADAMOU BOUBACAR
Guillaume LE GOFF

9 janvier 2022

Table des matières

Première partie

Description du projet

Le 2 juin 1999, l'aventurier Mike Horn part de la ville de Libreville, au Gabon. Il a pour objectif de faire le tour de la terre en suivant la ligne de l'équateur. Pour ce faire, sa première étape est la traversée de l'océan Atlantique sur son petit trimaran. Afin de ne pas perdre un temps précieux lors de ses phases de sommeil, il utilise un pilote automatique de barre franche pour continuer sa navigation.

Dans le cadre du "BE Synthèse et mise en œuvre des systèmes", il nous a été proposé de réaliser un pilote automatique de barre franche. Le travail nous a été facilité car le cahier des charges et les spécifications ont été écrites au préalable, nous permettant d'être clairement informés sur les fonctions à réaliser.

Les fonctions peuvent être placée dans 2 catégories : mesure de variables et commandes :

COMPAS : un capteur qui s'apparente à une boussole, envoie un signal au FPGA pour renseigner le cap. Ce signal s'apparente à un signal PWM, car la durée à l'état '1' permet d'accéder directement à la valeur du cap (entre 1° et 360°) par une simple loi de conversion.

GIROUETTE : le rôle de la girouette est de relever la direction du vent apparent [1]. Son principe de fonctionnement est similaire à celui du COMPAS.

ANEMOMETRE : l'anémomètre permet, quant à lui, de mesure la force du vent. Il fonctionne avec une modulation de fréquence d'impulsion. La loi de conversion $Hz \longrightarrow vitesse$ est très simple : une fréquence de f Hertz correspond à une vitesse de f km/h. L'étendue des mesures se fait entre 0 et 250Hz.

BOUTONS : Le barreur automatique possède un certain nombre de boutons poussoirs qui permettent de régler le cap souhaité, entre autre.

BUS AVALON : Il va nous permettre de pouvoir faire communiquer le micro-contrôleur avec ses périphériques UART.

Tout le contenu du projet est disponible sur GitHub en flashant le code ci-dessous ou à partir du lien à partir du lien du dépot [2]

1. Le vent apparent est constitué de la somme vectorielle de la direction du vent réel, et de la direction du vent relatif (créé par le déplacement).
2. Nous vous invitons à lire le REAME.md afin de mieux comprendre la disposition du répertoire GitHub

Deuxième partie

Mise œuvre des fonctions

1 Modules de base

Une bonne partie de nos fonctions reposent sur l'utilisation de composants similaires. Afin de réduire la complexité de notre code, nous avons opté pour la modélisation de composants génériques. Le premier était le composant COMPTEUR. Une schématisation logique est représentée FIGURE 1.

Le COMPTEUR est composé de quatre entrées et d'une sortie qui sont :

- `ARst` : *Asynchronous Reset*, sert à la remise à 0 du compteur à n'importe quel moment et n'est pas dépendant de l'horloge. Ce mode, bien que délicat à utiliser (puisqu'on essaie de garder un comportement synchronne avec l'ensemble de nos composants) trouve son utilité lors d'une remise à 0 "physique", comme l'appui sur un bouton.
- `SRrt` : *Synchronous Reset*, il a le même comportement que le `ARst` à l'exception qu'il agit de façon synchrone, sur front montant d'horloge. Il est également, de part la structure de son code, moins prioritaire son analogue asynchrone.
- `En` : *Enable*, il active le compteur pour qu'il puisse incrémenter sur front montant d'horloge.
- `Clk` : *Clock* l'entrée du signal d'horloge qui sert à cadencer le fonctionnement du compteur sur front montant.
- `Q` : il s'agit de la sortie du compteur.

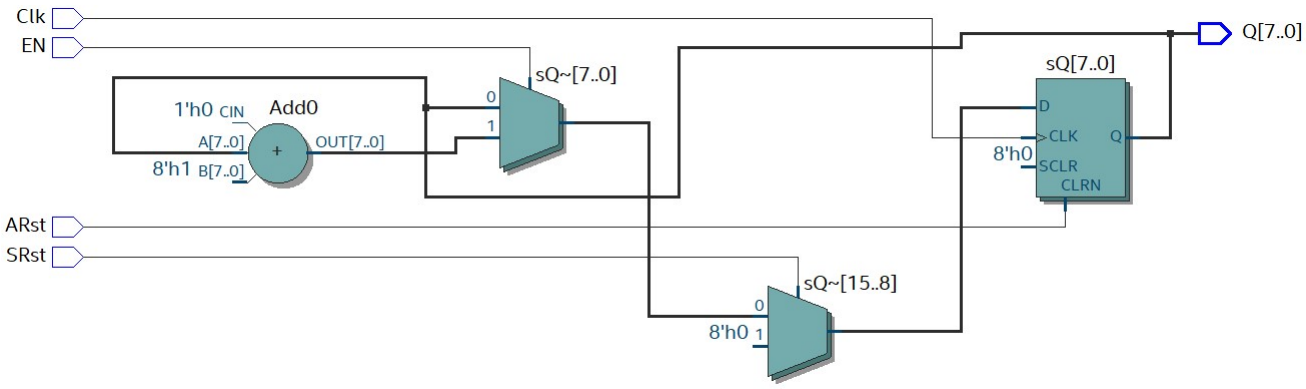

FIGURE 1 – Schéma logique du composant `Cnt` (*Compteur*)

```
1  -- Code de l'architecture principale du bloc fonction Cnt
2
3  pCnt : process(Clk, ARst)
4      begin
5          if ARst = '1' then sQ <= (others => '0') ;
6          elsif (Clk'event and Clk='1') then
7                  if SRst = '1' then sQ <= (others => '0') ;
8                  elsif EN = '1' then sQ <= sQ + 1 ;
9                  end if ;
10         end if ;
11     end process pCnt ;
12
13 Q <= sQ ;
```

La FIGURE 2 représente la simulation fonctionnelle du composant `Cnt`. Nous pouvons voir que l'entrée du reset asynchrone réinitialiser le compteur quelque soit l'état du cycle de l'horloge. De la même manière, nous pouvons constater que le passage a `'1'` de l'entrée reset synchrone ne réinitialise le composant que s'il coïncide avec un front montant d'horloge. Cela permet de ne pas dé-synchroniser le compteur de l'ensemble du circuit dans lequel il est utilisé.

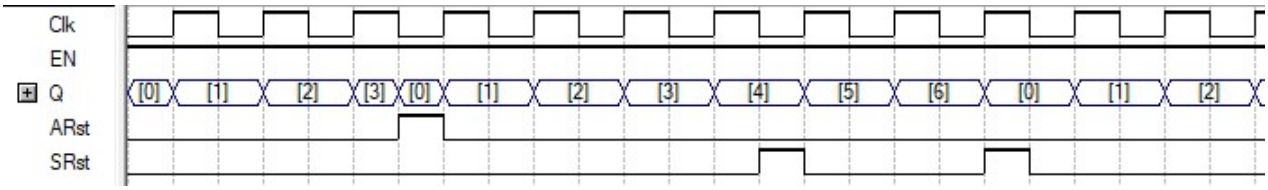

FIGURE 2 – Simulation fonctionnelle du composant `Cnt`

Par la suite, l'utilisation d'un détecteur de front montant s'est avéré indispensable. Nous avons donc créé un détecteur de toutes pièces en faisant le choix de ne pas trop le généraliser puisqu'il ne fait exclusivement que de la détection de front montant et pas de front descendant. Un schéma de câblage du composant est proposé dans la FIGURE 3.

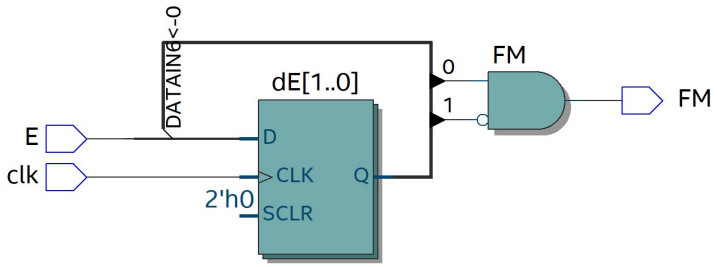

FIGURE 3 – Schéma logique du composant `detect_fm`

```
1  -- Code l'architecture principale du bloc fonction dectec_fm
2
3  signal dE : std_logic_vector(1 downto 0) ;
4  begin
5  bascule_E : process(clk, E)
6          begin
7                  if (clk'event and clk='1') then dE(0) <= E ;
8                          dE(1) <= dE(0) ;
9                  end if ;
10         end process bascule_E ;
11 FM <= '1' when dE(0) = '1' and dE(1) = '0' else '0' ;
12 end detection ;
```

Le composant `detect_fm` est composé d'une bascule synchrone sur le front montant d'horloge. Dans notre code initial, nous n'avions fait qu'une balance simple, mais nous avons été confrontés à des instabilités. Sur les conseils de M CARVALHO MENDES, nous avons opté pour le stockage du front montant dans un tableau de deux valeurs afin de limiter ces instabilités. Le composant est alors devenu très stable et maintien le signal de front montant pendant un cycle d'horloge (voir FIGURE 4). Ses entrées/sortie sont très simple, il n'y a qu'un signal d'entrée, synchronisé avec l'horloge interne et une sortie qui permet de récupérer l'information de front montant.

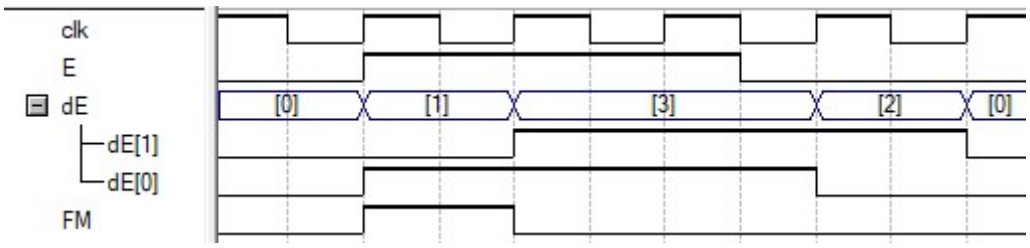

FIGURE 4 – Simulation fonctionnelle du composant `detect_FM`

Note : Une adaptation de ce composant est possible et ne nécessite que très peu de modification afin d'obtenir son analogue de détection de front descendant.

2 Module COMPAS

Le premier composant, disons "plus haut niveau", que nous avons réalisé est le COMPAS. Son rôle est expliqué au début de ce rapport, en introduction. Son fonctionnement est relativement simple, il repose sur le temps que passe le signal d'entrée à l'état haut. La loi de conversion est directe : $10ms$ à l'état haut représente une incrémentation de 1° de la mesure du cap. Une schématisation de notre module est représenté par la FIGURE 5.

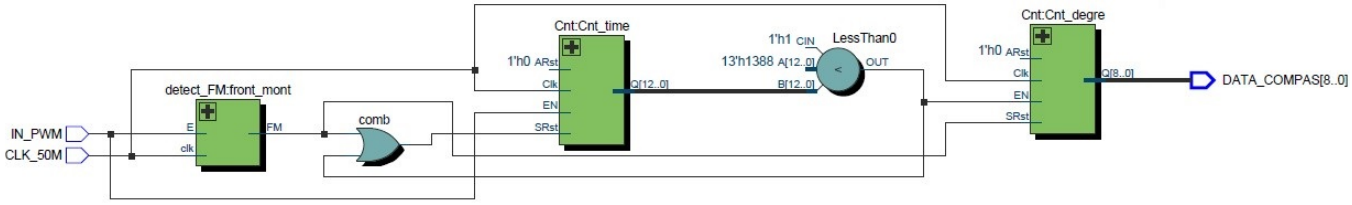

FIGURE 5 – Schéma logique du composant `compas`

Dans sa composition, le module COMPAS possède deux compteur identiques. Le premier compteur (que nous avons nommés `Cnt_time`) est associé à un comparateur qui compare la valeur de sortie à la valeur 5000 [3]. Cette sortie de compteur est ensuite reliée à l'entrée `En` d'un autre compteur (`Cnt_degre`) qui s'incrémentera alors de 1° tous les activation de son entrée.

Cette fonction a été la première véritable fonction prenant en compte des composants reliés par des `port map`. Nous avons passé beaucoup de temps a tenter de comprendre les erreurs de syntaxe et les erreurs de logique que nous renvoyait le compilateur. Ce temps passé n'a cependant pas été vaint puisqu'il nous a été nécessaire pour appréhender toute la logique d'un langage de programmation concurrent, chose que nous n'avions pas l'habitude puisque nous avons une grosse formation en C qui est un langage de programmation séquentiel.

3 Module GIROUETTE

Le module que nous avons réalisé, par la suite, est la GIROUETTE. Son fonctionnement est identique à celui du COMPAS, donc, sa réalisation est également très proche. C'est pour cela que sa réalisation s'est faite très rapidement, ainsi que sa validation. A ce stade, nous avons fait le choix de ne pas représenter la schématisation *single-shot* et *continuous* afin de ne pas surcharger le schéma. Dans l'idée, le fonctionnement sera une machine à état qui agira sur l'activation des compteurs [4].

3. Pour une horloge de $50MHz$, une variation de $10ms$ correspond à 5000 fronts montants d'horloge.
4. Pour plus de détails, nous vous invitons à regarder le module ANEMOMETRE.

4 Module ANEMOMETRE

4.1 Module ANEMOMETRE "simple"

Le fonctionnement de l'anémomètre repose sur un train d'impulsion envoyé par le capteur. La fréquence des front montant permet d'accéder à la vitesse par une conversion directe, c'est-à-dire que $1s^{-1} \rightarrow 1km/h$. Pour réaliser la fonction, il nous a fallut un compteur battant la seconde, puis en cascade, un autre compteur relié au détecteur de front montant. Tant que le compteur de 1 seconde n'est pas arrivé à son terme, le seconde compteur s'incrémente au rythme de l'arrivée des fronts montants, jusqu'à ce que le premier compteur atteigne 1 seconde. A ce moment, la valeur de la vitesse qui était stockée jusqu'à présent dans une variable tampon est copiée sur la sortie du module. De cette façon, nous avons toujours une valeur stable et facilement lisible, en revanche, la donnée n'est rafraîchie que toute les secondes (voir la variable `vitesse` sur la simulation fonctionnelle FIGURE 6).

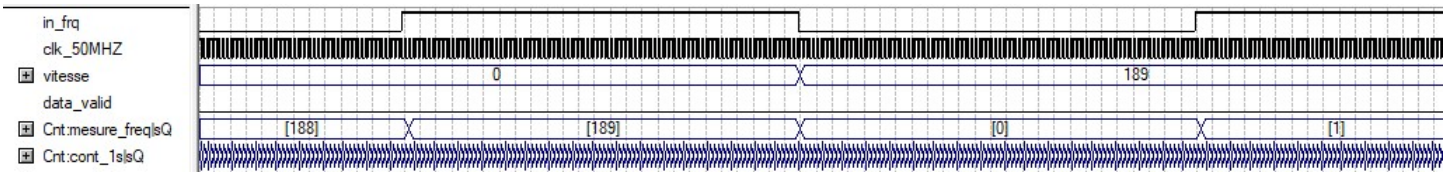

FIGURE 6 – Simulation fonctionnelle du composant `anemometre`

Dans notre phase de test, nous avions réussi "par chance" à synchroniser notre générateur de signal avec un front montant d'horloge. Nous avons donc validé le bon fonctionnement de l'anémomètre et nous sommes passés à la suite. Or, le composant qui fonctionnait parfaitement il y a quelques heures se retrouvait à présent avec un comportement de type "guirlande de Noël" [5]... Nous avons mit beaucoup de temps à trouver notre erreur, nous avons revu en détail notre schéma fonctionnel, notre code VHDL, nos conventions de logique normale et logique inverse, sans succès. Ce n'est qu'avec l'aide de l'outil *Signal Tap Logic Anlyzer* proposé par *Quartus* que nous avons eu une piste. Concrètement, nous avons appliqué une fréquence de $38Hz$ à notre port d'entrée et nous relevions des valeurs oscillant autour de $38Hz$ sans jamais s'arrêter sur une valeur fixe (vraie ou fausse) qui aurait pu nous aiguiller. La réponse à notre problème est assez sournoise : si notre composant `detect_FM` était bel et bien synchrone avec le reste de notre système, le signal PWM d'entrée, agissant également sur le `reset` du compteur 1 seconde n'était pas synchronisé. Ce petit artefact avait pour conséquence d'effectuer des reset anticipés, manquant ainsi la détection de quelques front montants [6].

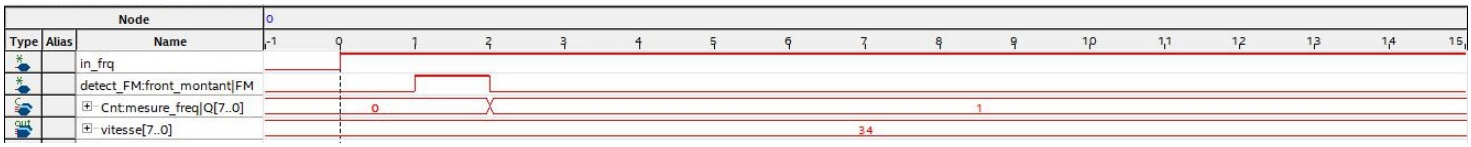

FIGURE 7 – Relevé de quelques signaux pour une vitesse de $34km/h$

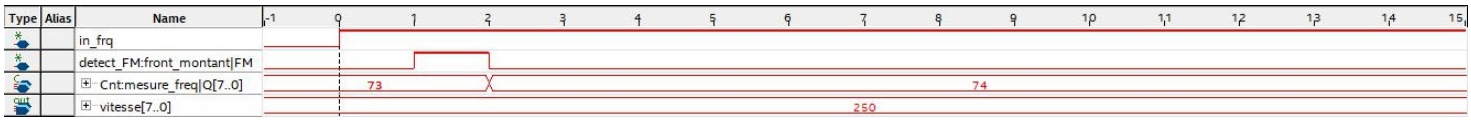

FIGURE 8 – Relevé de quelques signaux pour une vitesse de $250km/h$

5. Nous simulions le plus possible sur la carte de développement, en utilisant les LED, d'où le côté guirlande.
6. Le code originel, et donc partiellement défaillant est disponible dans le dossier GitHub au répertoire `[PROJET]Module_anemometre`

Les FIGURE 7 et FIGURE 8 illustrent la capture d'une vitesse de $34km/h$ et de $250km/h$ (vitesse max), respectivement. Nous pouvons observer le déclenchement du composant `Cnt:mesure_freq` sur détection du front montant sur l'entrée `in_frq`.

Sur la FIGURE 9 est représenté une schématisation du composant `anemometre` dans sa version 2 (c'est-à-dire fonctionnelle et stable) [7]. Nous pouvons clairement identifier la bascule servant à synchroniser le signal d'entrée avec le front mont montant de l'horloge interne du système. Nous visualisons également, en toute fin, la bascule permettant l'écriture de la vitesse mesurée sur la sortie.

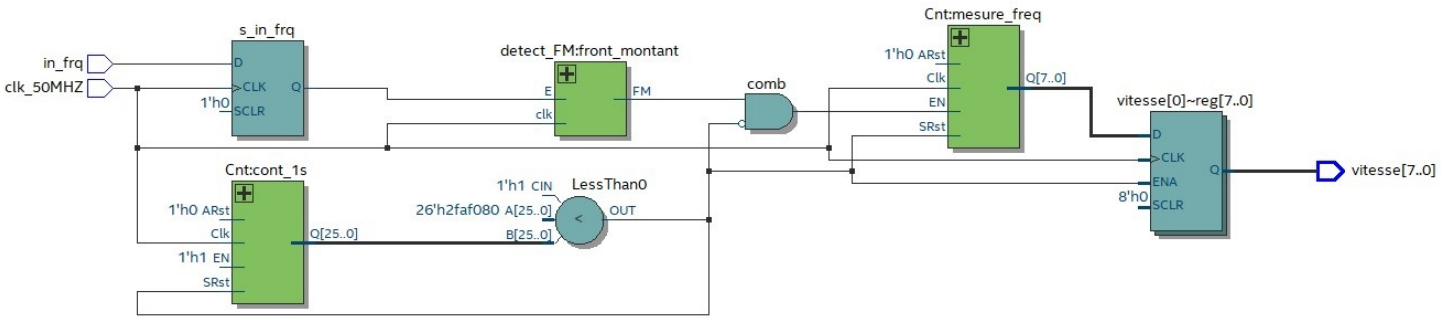

FIGURE 9 – Schéma logique du composant `anemometre`

4.2 Module ANEMOMETRE "multi-modes"

L'étape suivante, est de complexifier un peu nos fonctions afin d'y incorporer des modes de mesures. Parmi ces modes, il y a le mode *continuous* et son complémentaire *single-shot*. Il y également l'ajout d'un véritable "bouton" *raz*. Le schéma de ce composant amélioré est représenté en FIGURE 10. Comme nous pouvons le constater, il s'agit du même composant que celui de la FIGURE 9 à la différence près qu'il comporte une machine à états (partie la plus à gauche du schéma).

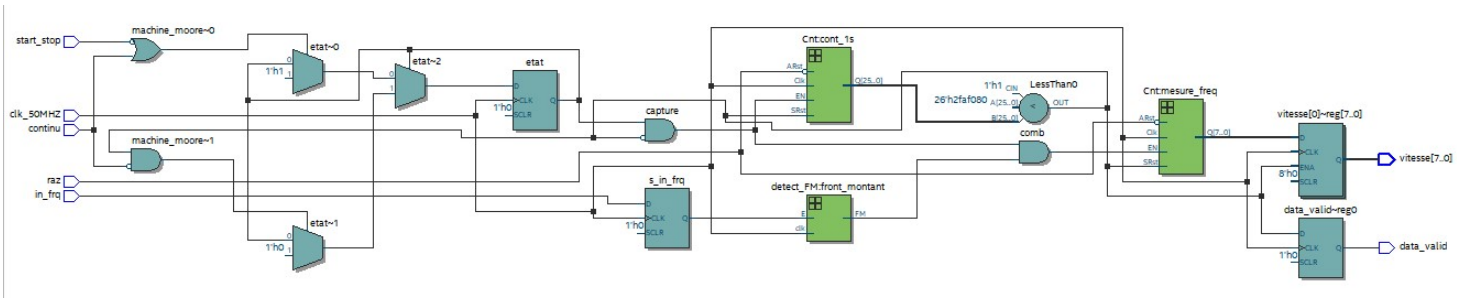

FIGURE 10 – Schéma logique du composant `anemometre_modes`

Cette machine à états (`etat`) pilote la valeur d'un signal d'entrée (signal `capture`), connecté aux broches *Enable* des compteurs. Le signal capture est de type booléen, prenant tantôt la valeur `'1'` pour démarrer une acquisition, tantôt la valeur `'0'` pour bloquer l'acquisition.

7. Module anémomètre dans sa version 2 disponible dans le répertoire `[PROJET]Module_anemometre_v2`

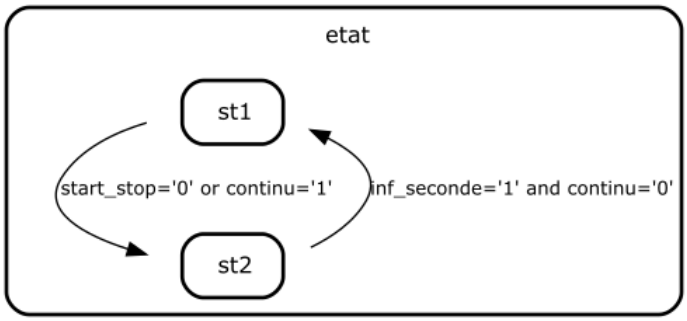

FIGURE 11 – Machine a état permettant le fonctionnement des modes

La machine a état peut donc être dans 2 états (as simultanément, bien entendu), mais la capture de la vitesse ne se faire que lorsqu'elle est dans l'état 2 (`st2`). On peut alors dire qu'il existe un état de repos (`st1`) et un état de capture (`st2`). En effet, une capture (ou mesure de la vitesse) ne se fera uniquement que si le mode continu est activé (`continu='1'`) OU si l'action de capture est demandée par appui sur le bouton `start_stop`.

Par ailleurs l'entrée `start_stop` est en logique inversée, c'est-à-dire qu'elle est active à l'état bas, cela est dû au fonctionnement physique des boutons de la carte de développement DE0-nano. Un retour à l'état de repos n'est possible que si la mesure a été effectuée (`inf_seconde='1'`)[8] ET que le mode continu n'est pas activé, le cas échéant, nous restons dans l'état de capture, le signal `inf_seconde` est réinitialisé et nous reprenons un nouveau cycle de mesure.

L'extrait de code ci-dessous représente la machine a états permettant la navigation entre les deux modes d'acquisitions proposés.

```
1  -- Code de la machine a etats des 2 modes
2  machine_moore : process(clk_50MHz)
3      begin
4      if(clk_50MHz'event and clk_50MHz='1') then
5          case etat is
6              when st1 => if(start_stop='0' or continu='1') then
7                      etat <= st2 ;
8                  end if ;
9              when st2 => if(inf_seconde='1' and continu='0') then
10                     etat <= st1 ;
11                 end if ;
12         end case ;
13     end if ;
14 end process ;
```

8. Remarque : Nous aurions également pu mettre le signal `data_valid` à la place puisqu'ils ont le même comportement.

5 Module de gestion des boutons

5.1 Description

L'objectif de ce module est d'assurer le fonctionnement du clavier du pilote automatique. En effet,ce dernier est composé de cinq touches (bâbord, tribord, stby, nav, tack) qui permettent d'ajuster le cap avec précision et d'utiliser des fonctions de navigation. Ainsi, après un appui simple sur le bouton bâbord ou tribord, on modifie le cap de 1° selon la direction choisie, et de 10° lorsqu'on exerce une pression prolongée.

Le fonctionnement de ce système est assuré en trois mode :
— le mode VEILLE : c'est ce mode qui est activé dès la mise en marche de l'appareil.
— le mode MANUEL : ce mode consiste à actionner le vérin en le faisant rentrer ou sortir lorsque l'appareil est en mode veille.
— le mode AUTOMATIQUE : ce mode est activé lorsque le bouton Stand-By/Auto est appuyé. Il permet de verrouiller le pilote sur un cap choisi.

La prise en compte de tous ces réglages par le pilote est confirmé par des "bip" sonore et des éclats de LED. Pour la mise en œuvre de ce module, nous avons choisi de le subdiviser en plusieurs sous blocs afin de prendre en compte son fonctionnement intégral. Ainsi nous avons les différents sous bloc suivants : machine à état, leds, bip, temporisation, clk_100hz, clk_50hz et clk_1hz.

5.2 Machine à état

Ce bloc prend en compte les différents mode de fonctionnement du système et les différents code de fonction proportionnels aux treize états qui le compose. La FIGURE 12 représente la machine à état associée.

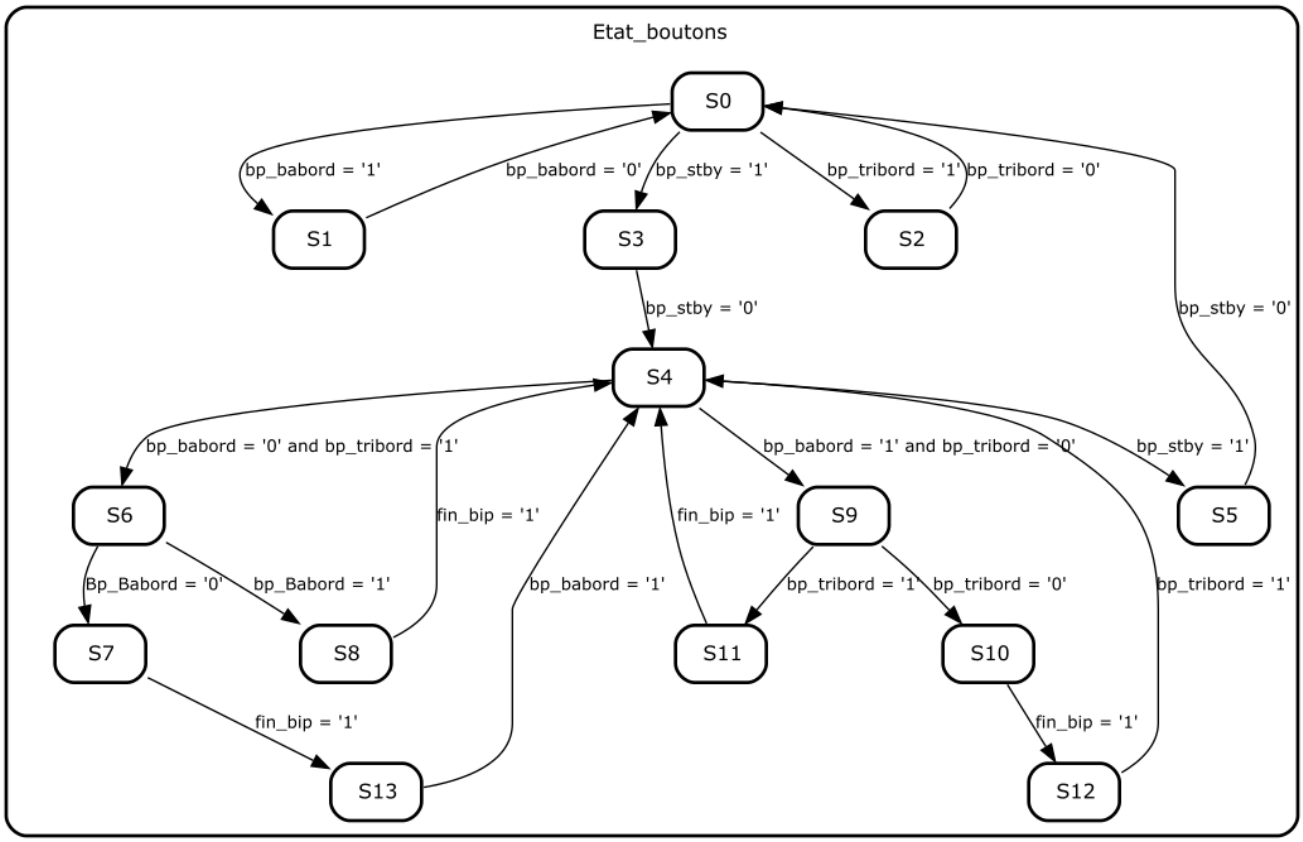

FIGURE 12 – Machine a état de la gestion des boutons poussoirs

5.3 Module BIP

Ce bloc permet de générer des bips simples ou doubles selon l'appui (appui simple ou prolongé). Le bip simple correspond à une incrémentation de 1° suite à un appui simple. Le double bip correspond à une incrémentation de 10° suite à une appui double . Pour sa mise en uvre nous avons utilisé une machine à état (FIGURE 13) qui prend en compte les trois différents états de fonctionnement : veille (pas de bip), simple et double.

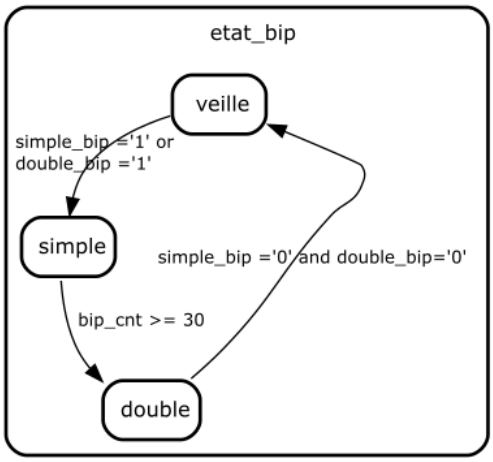

FIGURE 13 – Machine à état de la gestion des bip

5.4 Module LED

Pour assurer le fonctionnement des leds, nous avons généré un compteur de sept bits dont les valeurs en sortie sont proportionnelles à l'intensité des leds. Nous avons deux niveaux d'intensité : faible et intense. Un extrait du code VHDL associé au module LED est représenté ci-dessous.

```
1  led: process(clk_100,raz)
2  begin
3      if (raz = '0') then
4          cmpt <= 0;
5          intensite <= (others =>'0');
6      elsif rising_edge(clk_100) then cmpt <= cmpt + 1;
7          if(cmpt = 6) then cmpt <=0;
8          end if;
9          case cmpt is
10             when 0 => intensite <="10000000";
11             when 1 => intensite <="11000000";
12             when 2 => intensite <="11100000";
13             when 3 => intensite <="11110000";
14             when 4 => intensite <="11111000";
15             when 5 => intensite <="11111100";
16             when 6 => intensite <="11111110";
17             when others => intensite <= (others =>'0');
18         end case;
19     end if;
20     led_faible <= intensite(1);
21     led_intense <= intensite(6);
22 end process led;
```

6 Bus Avalon

Une application codée en VHDL peut parfois montrer des limites. En effet, le langage ne permet pas l'utilisation temps réel de l'application en console, notamment à cause de l'absence de fonction *print*. Pour cela, il est possible de "transformer" notre FPGA en une sorte de microcontrôleur, qui sera, lui, programmable en C/C++ !

Cette transformation, ou plutôt, devrait-on parler de configuration, de notre FPGA en microcontrôleur se fait *via* l'implémentation d'une architecture de mémoire (RAM), du choix d'un type de processeur (ici le processeur NIOS) et des périphériques d'entrée/sortie (UART) pour l'envoi et la réception des différents registres.

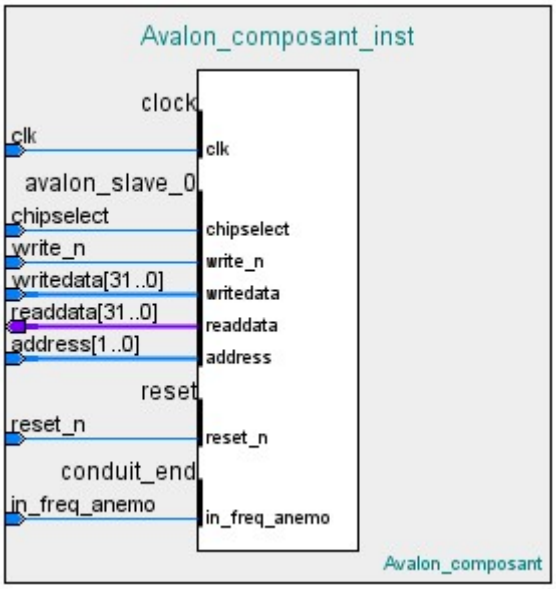

FIGURE 14 – Shéma du composant bus Avalon

A ce périphérique, nous y avons greffé un bus sous forme de composant : il s'agit du bus Avalon (voir schématisation FIGURE 14). Ce composant est décrit sous une forme de code VHDL [9] et intègre les modules précedemment codés sous forme de composant (grâce à la fonction `port map();`) [10]. Le mappage du bus comprend :

- une broche `clk` qui correspond à l'horloge
- une borhce `write_n` qui correspond à l'état d'écriture (en logique inverse : `'0'` signifie que le registre est en écriture
- une broche `reset_n` remise à zéro des données contenues dans les registres (en logique inverse)
- `writedata` et `readdata` qui correspondent aux demandes d'écriture et de lecture sur le registre, respectivement.
- la broche `address` permet de relier le bus à l'emplacement de la mémoire. L'adresse est laissée à la discrétion de l'outil de configuration du NIOS :*Platform Designer*.

A présent que nous avons configuré le micro-contrôleur, il ne reste plus qu'a le programmer en C. Pour cela, nous avons utilisé le logiciel *Nios II Software Build Tools*, proposé par *Quartus* et basé sur l'environnement *Eclipse*.

9. Module Avalon disponible dans le répertoire `[PROJET]NIOS/Nios_logique.vhd`
10. Remarque : Ici nous n'avons ajouté que le composant anemomètre mais il est facile d'en ajouter plusieurs avec l'utilisation des `port map();`

```c
/* Fonction d'acquisition de l'anemometre */
#include "stdio.h"
#include "system.h"
#include "unistd.h"
#define continu (volatile unsigned int*)AVALON_COMPOSANT_0_BASE
#define data (volatile unsigned int*)(AVALON_COMPOSANT_0_BASE + 4)

void get_anemo(){
                *continu = 0x3 ;
                int anemo_data = *data ;
                printf("%d\n", anemo_data & 0xFF) ;
                usleep(100000) ;
}
```

Le code C ci-dessus est celui utilisé dans la fonction d'acquisition de la vitesse du vent. Nous avons, en premier lieu, configuré la variable `*continu` par un masquage par la valeur `0011`, ce qui correspond à la mise à 1 des signaux `Ni_continu` et `Ni_start_st` [11]. Cette variable est définie à partir de la variable `AVALON_COMPOSANT_0_BASE`, contenue dans la bibliothèque `system.h`. La variable `data` est également définie par rapport à `AVALON_COMPOSANT_0_BASE` avec un masquage en OU sur le bit n° 3, ce qui correspond au bloquage de la remise à zéro [12]. Enfin, nous affichons la valeur du registre en appliquant un masque en ET (`"0xFF"`) sur les 32 bits qui composent le registre `readdata` [13]. Ce registre est composé (du bit de poids faible vers le bit de poids fort) : des 8 bits de la vitesse mesurée et du bit de validité de la mesure (`data_valid`), auxquels sont rajoutés 23 bits à l'état `'0'`, afin d'obtenir un registre sur 32 bits.

Dans le code C principal, nous ajoutons la librairie contenant la fonction précédemment décrite dans la zone de déclaration globale, puis nous l'appelons tout simplement par `get_anemo();` en prenant le soin de laisser quelques instants pour que l'affichage se fasse sur la console et que nous ne soyons pas noyés sous un flot de valeurs avec `usleep();`.

Par un manque de temps dû aux différents problèmes que nous avons rencontrés, nous n'avons pas pu incorporer toutes les fonctions décrites dans ce rapport dans le bus Avalon, cependant, la démarche reste sensiblement la même.

11. Voir le code `[PROJET]NIOS/Nios_logique.vhd`, lignes 58 et 59
12. Signal `Ni_raz`, voir le code `[PROJET]NIOS/Nios_logique.vhd`, ligne 57
13. Voir le code `[PROJET]NIOS/Nios_logique.vhd`, ligne 65

Troisième partie

Conclusion

Nous entendons parler du langage VHDL, dans notre cursus, depuis maintenant plusieurs années. Bien que nous l'ayons approché en Licence, nous n'avons jamais eu de réels cours avant cette année. Comme nous le disions dans ce rapport, la démarche intellectuelle à laquelle nous sommes habitués pour programmer ne s'applique pas totalement avec la manière de faire du VHDL. La première différence est que le langage est, par défaut, concurrent, à moins que nous "arrêtions le temps" en lançant un processus. Là encore, nous avons eu du mal à bien cerner l'utilité et la déclaration des signaux de sensibilité des processus. La gestion des phénomènes temporels ont aussi été sources de problèmes tout au long de notre projet. En effet, nous manipulions des données cadencées par une horloge (ici en l'occurrence, 50 Hertz), il fallait donc prendre soin de synchroniser nos opérations le plus possible afin de minimiser les temps de propagation ou la prise en compte de *glitch* qui auraient pour conséquence de déstabiliser notre système.

Dans un contexte de mise en œuvre, nous avons du faire face aux aléas de la pratique. Effectivement, bien que notre code fonctionne comme nous le souhaitons, de façon théorique et à partir des simulations réalisées, le passage dans le monde réel peut parfois faire apparaître des erreurs que nous n'aurions pas anticipées, nous obligeant à revoir notre code afin d'assurer une bonne stabilité. Ce genre de péripétie nous a fait perdre beaucoup de temps et il aurait pu être réduit si nous avions davantage sollicité le coup de pouce de l'enseignant.

Bien que nous n'avons pas pu aller aussi loin que nous le souhaitions, au regard du chemin que nous avons parcourus, nous sommes fier du travail effectué.

Nous remercions Monsieur Pedro CARVALHO MENDES pour son aide précieuse apportée durant ce projet ainsi que pour ses conseils tout au long du semestre.

* * *