

Systèmes interconnectés – SINT

Sujet de TP 5 : CAN et FPGA

Centrale Nantes

P.-E. Hladik, pehladik@ec-nantes.fr

—

Version bêta (27 novembre 2023)

1 Objectifs pédagogiques

- spécifier une machine séquentielle à l'aide d'un automate
- simuler et tester une implémentation en VHDL
- intégrer plusieurs entités dans un projet
- comprendre le calcul du CRC

2 Rendu

À la fin du TP déposez sur Hippocampus une archive avec un compte rendu en pdf (voir les éléments demandés dans le texte).

3 Prise en main

Cette première partie a pour but de reprendre en main Vivado et le VHDL.

3.1 Création d'un projet

Commencez par créer un nouveau projet sous Vivado (voir le [document en ligne](#) ou la vidéo). Pour cette première étape vous aurez besoin des fichiers `clock_div.vhd` en tant que *Design Sources*, de `testbench_div.vhd` comme *Simulation Sources* et `basys3.xdc` comme contraintes. Ces fichiers sont dans l'archive sur Hippocampus. Vous pouvez importer les fichiers à la création du projet ou bien après.

Pour la carte Basys3, il faut sélectionner le composant « xc7a35tcpg236-1 » que l'on peut trouver avec le filtre :

- Family : Artix-7
- Package : cpg236
- Speed : -1

3.2 Simulation

Vérifiez que vous avez bien importé `testbench_div.vhd` dans *Simulation Sources* et qu'il est bien au *top level* (sinon clic droit et *Set as Top*). Ce fichier simule le comportement d'une entité de type `clock_div`.

Lancez la simulation comportementale (voir vidéo) et observez. Pour l'instant seul le signal `reset` est simulé. Ajoutez le code nécessaire pour simuler l'horloge à une période de 10 ns et lancez la simulation.

Rapport : mettez l'extrait de code du testbench ainsi qu'une capture de la simulation.

Quel est le rapport cyclique si le diviseur est impair (3 par exemple) ? Observez en simulation le comportement et expliquez les raisons en allant lire le code de `clock_div.vhd`.

Rapport : mettez une capture de la simulation avec un diviseur de 3 et expliquez le comportement observé à partir du code.

3.3 Synthèse

Assurez-vous que vous avez bien importé `display_sint.vhd` comme *Design Sources* et `basys3.xdc` comme contraintes.

Créez un nouveau fichier de conception et mettez le comme *Top Level*. Implémentez un système qui fait clignoter une led toutes les 500 ms. Pour cela utilisez le diviseur d'horloge sachant que l'horloge du système a une période de 10 ns. Les signaux physiques dont vous aurez besoin sont :

- `led0` en sortie,
- `reset` en entrée,
- `clock_100Mhz` en entrée.

Ils doivent être déclarés comme ports de votre entité comme dans l'exemple ci-dessous :

```
entity blink is
  Port (
    reset          : in std_logic;
    clock_100Mhz   : in std_logic;
    led0           : out std_logic
  );
end blink;
```

et seront ensuite liés aux signaux physiques dans le fichiers de contraintes (vérifiez que les lignes nécessaires dans le fichiers de contraintes `basys.xdc` sont bien décommentées).

Ecrivez le code, faites la synthèse et exécutez le résultat sur la carte.

Rapport : mettez le code produit.

4 Affichage

Importez dans votre projet le fichier `display_sint.vhd`.

L'entité `display_basys3` permet d'afficher une valeur en hexadécimale sur l'afficheur 7 segments de la carte. Les ports sont :

- `clock_100Mhz` (entrée) : l'horloge physique de la carte,
- `reset` (entrée) : le signal physique de reset de la carte,
- `anode_activate` (entrée) : les signaux physiques des anodes,
- `led_out` (entrée) : les signaux physiques des leds de l'afficheur,
- `displayed_number` (entrée) : la valeur à afficher.

Créez un nouveau fichier et implanter un code qui met à jour toutes les secondes l'afficheur en incrémentant une valeur (vous aurez une horloge hexadécimale). Servez-vous de ce que vous avez fait avant pour le compteur en le copiant dans le nouveau fichier (ou utilisez ce que vous avez fait avant).

Si vous passez par un entier pour coder la valeur à afficher, vous pouvez utiliser

```
n <= std_logic_vector( to_unsigned(val, n'length))
```

pour convertir l'entier non signé `val` et l'affecte à un signal `n` de type `std_logic_vector` de taille `n'length`.

Pour les contraintes, voir les commentaires dans le fichier `display_sint.vhd` et décommentez les lignes dans le fichier de contraintes.

Rapport : mettez le code produit.

5 Capteur ultrason

Dans cette partie vous allez coder le contrôleur d'un capteur **Ultrasonic Ranger V2.0** de Grove. Le principe de fonctionnement d'un capteur à ultrason consiste à émettre « à intervalles réguliers de courtes impulsions sonores à haute fréquence. Ces impulsions se propagent dans l'air à la vitesse du son. Lorsqu'elles rencontrent un objet, elles se réfléchissent et reviennent sous forme d'écho au capteur. [Le capteur] calcule alors la distance le séparant de la cible sur la base du temps écoulé entre l'émission du signal et la réception de l'écho. » [[microsonic](#)].

Les entrées/sorties de l'ultrasonic ranger sont décrites par la figure ?? . Le seul signal qui nous intéresse est celui identifié comme **SIG**. Physiquement vous pouvez connecter le capteur sur les broches des pmod de la basys3 (voir figure ??).

5.1 Spécifications

L'entité permettant de contrôler le capteur à ultrason aura pour interface les ports suivants (voir figure ??) :

- en entrée :
 - `clk` : une horloge,
 - `rst` : un signal de reset,
 - `start` : un signal pour démarrer une acquisition.
- en sortie :

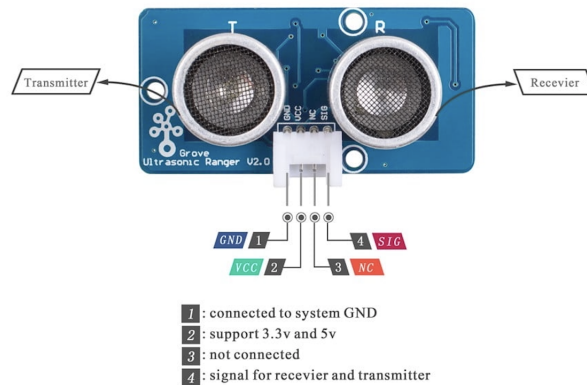


FIG. 1 – Entrée/sortie de l'Ultrasonic Ranger V2.0

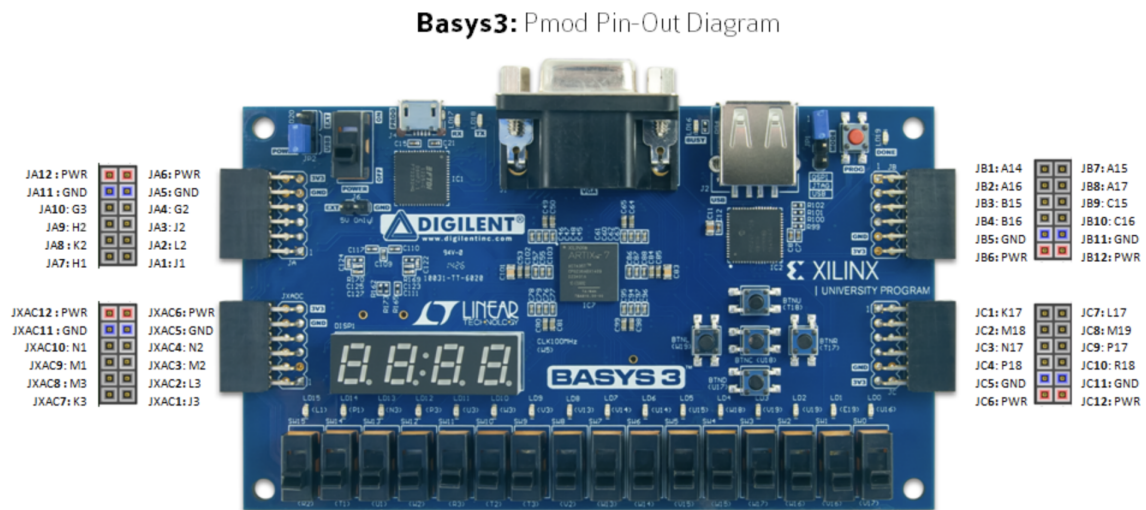


FIG. 2 – Nom des broches des pmod de la basys3

- ready : un signal indiquant que le capteur est disponible pour une mesure,
- resReady : un signal indiquant qu'une nouvelle valeur a été produite,
- res : un entier avec la distance mesurée en cm,
- err1 : un signal en cas d'erreur de type 1 (voir après),
- err2 : un signal en cas d'erreur de type 2 (voir après).
- en entrée/sortie (inout en VHDL) :
 - sig : le signal de contrôle et de donnée vers le capteur externe.

L'entité aura aussi deux valeurs de configuration `timeout1` et `timeout2`.

Le fonctionnement du capteur est simple. Pour lancer une mesure, il faut maintenir sur `sig` un signal à l'état haut pendant $10 \mu s$, puis attendre de nouveau le passage à l'état haut du signal pour commencer la mesure et arrêter la mesure quand le signal repasse à l'état bas (voir figure ??). La distance à l'obstacle est proportionnelle à la durée de l'impulsion mesurée.

Le contrôle à réaliser est donc le suivant :

- Le signal `ready` est à 1 quand le capteur est au repos et 0 sinon.

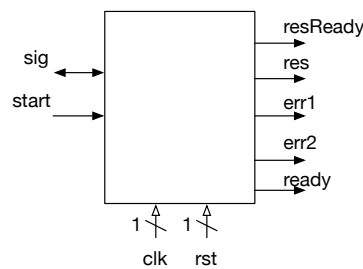


FIG. 3 – Port de l'entité ultrason

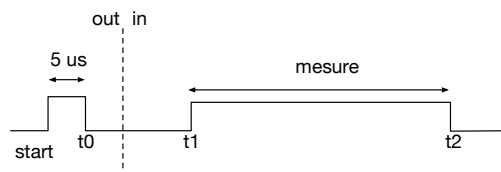


FIG. 4 – Signal de contrôle et de mesure du capteur

- Quand le signal **start** est à 1 et que **ready** est à 1, une impulsion de $10\ \mu\text{s}$ est produite, puis le signal est placé dans l'état Z pour permettre la mesure.
- La mesure est le temps de l'impulsion. Pour une impulsion dt en μs , la distance est de $dt/29/2\ \text{cm}$.
- Quand une mesure est produite, **resReady** est mis à 1 et le résultat (en cm) est écrit dans **res**.
- Si le temps entre **t0** et **t1** est supérieur à **timeout1** alors une erreur de type 1 est produite (le signal **err1** passe à 1).
- Si le temps entre **t1** et **t2** est supérieur à **timeout2** alors une erreur de type 2 est produite (le signal **err2** passe à 1).
- les signaux de sortie sont tous à zéro au début et au lancement d'une nouvelle mesure.

Représentez la spécification du contrôleur ultrason sous forme d'un automate (Mealy ou Moore au choix).

Rapport : mettre l'automate qui décrit la spécification du contrôleur ultrason.

5.2 Implantation

Produisez le code de votre contrôleur ultrason.

Rapport : mettre le code du contrôleur ultrason.

5.3 Simulation

Faites une simulation de votre contrôleur. Pour cela vous aurez besoin de produire un signal d'horloge, mais aussi de **sig** (tant que le signal est contrôlé il faut qu'il soit dans l'état Z).

Rapport : mettre le code du testbench et une figure montrant son comportement.

5.4 Synthèse

Ecrivez un nouveau code utilisant votre contrôleur ultrason ainsi que l'afficheur pour que :

- une mesure soit faite toutes les 500 ms,
- le résultat soit affiché,
- la led0 clignote pour montrer une activité,
- la led1 s'allume s'il y a une erreur de type 1,
- la led2 s'allume s'il y a une erreur de type 2,
- la led3 s'allume quand il y a une mesure de réaliser.

Il est fortement conseillé de bien identifier vos entrées/sorties avant de commencer et de faire un petit automate avant de se lancer dans le code.

Rapport : mettre le code produit.

6 Communication via un bus CAN

Dans cette partie, vous allez utiliser la bibliothèque CanLite développée par B.G. Gardner et disponible sur GitHub <https://github.com/bggardner/can-lite-vhdl>. Le code propose une implémentation d'un contrôleur CAN en VHDL. Il a été produit à partir d'un projet de Igor Mohor disponible sur opencores.org. Les fichiers sources (`CanBus_pkg.vhd` et `CanLite.vhd`) sont aussi disponibles dans l'archive sur Hippocampus.

Nous n'utiliserons cette bibliothèque que pour l'envoi d'un message. La réception est opérationnelle mais ne sera pas traitée ici (il faut quand même cabler Rx pour que l'envoi d'un message soit validé).

Les ports du composant `CanLite` sont :

- `Clock` : pour l'horloge (voir le readme du dépôt pour configurer la vitesse de transmission du bus en fonction de la fréquence de l'horloge),
- `Reset_n` : pour ré-initialiser le contrôleur. Il faut qu'il soit à 1 pour que le bus fonctionne,
- `RxFram` : signal de type `CanBus.Frame` avec la valeur de la trame en réception,
- `RxFifoWriteEnable` : signal pour informer de l'arrivée d'un message,
- `RxFifoFull` : non identifié (a priori indique sur le buffer de réception est plein) ,
- `TxFram` : trame à envoyer,
- `TxFifoReadEnable` : signal à mettre à 1 pour initier l'envoi, puis à remettre à 0 pour éviter les envois multiples,
- `TxFifoEmpty` : non identifié (a priori indique sur le buffer d'envoi est plein),
- `TxAck` : Impulsion à l'état haut lorsqu'un message a été transmis avec succès,
- `Status` : état du bus,
- `CanRx` : ligne de réception à mapper sur une broche d'un Pmod,
- `CanTx` : ligne de transmission sur une broche d'un Pmod.

L'envoi d'un message se fait en écrivant dans `TxFram` la trame à envoyer, puis en validant l'envoi en mettant à l'état haut `TxFifoReadEnable` (attention s'il reste à l'état haut le message sera immédiatement renvoyé quand la première communication sera terminée).

6.1 Calcul du CRC

Observer le code dans `CanLite` et trouver la partie traitant le CRC. Expliquer le fonctionnement de l'algorithme pour calculer le CRC sachant que :

- les ports d'entrée `enable` et `initialize` servent uniquement à contrôler l'avancer du calcul,
- au $i^{\text{ème}}$ front montant de l'horloge `clk`, le port d'entrée `data` a pour valeur le $i^{\text{ème}}$ bit de la trame (en commençant par la valeur MSB).

Rapport : expliquez comment le calcul est fait.

6.2 Mise en œuvre d'une communication sur un bus CAN

Créer un nouveau code en vous inspirant de ce que vous avez fait avant et ajoutez l'envoi de la donnée via le bus CAN.

Rapport : réaliser un automate décrivant l'ensemble de l'application et mettez le code produit.