FPGA comme co-processeur de Xenomai/Linux temps-réel

G. Goavec-Merou, 16 décembre 2016

1 Objectifs

Nous avons vu que l'extension temps-réel de Linux – Xenomai – a pour vocation de réduire la variabilité des temps de réponse du système à une sollicitation, et en particulier de borner l'intervalle de temps entre l'évènement et l'action associée.

Pour évaluer les diverses solutions (temps-partagé, temps-réel et FPGA) nous allons utiliser une plateforme Armadeus Systems APF27 disposant d'un environnement logiciel où Xenomai est disponible et d'un FPGA.

Un avantage majeur de la combinaison CPU+FPGA est de fournir une hiérarchie de contrainte sur les lattences – faible sur le CPU, bonne sur l'extension Xenomai, excellente sur le FPGA – au détriment de la complexité de l'implémentation d'un algorithme donné – simple sur CPU, plus complexe sur Xenomai, long à déverminer sur FPGA en VHDL. La résolution temporelle des opérations sur le FPGA est de 10 ns, puisque le cœur du FPGA est cadencé à 100 MHz [1]

Le présent TP s'articule en deux grandes parties :

1. Réalisation d'applications en espace-utilisateur s'éxecutant dans le domaine temps-partagé et dans le domaine temps-réel en utilisant une mise en veille du processus ou une activation par timer. Ces applications devront générer un signal périodique sur une broche du processeur. Le FPGA, qui offre un aspect temps-réel matériel, sera donc exploité pour mesurer les fluctuations d'un signal généré à partir du processeur soit dans le domaine temps partagé (*Linux*), soit temps-réel (*Xenomai*). Un ensemble de durées doivent être stockées dans une mémoire interne du FPGA, puis transmises au processeur pour analyse par tracé de l'histogramme des intervalles de temps de commutation d'un GPIO.
2. Ensuite nous allons nous focaliser sur la partie FPGA en vu, de réaliser un compteur de période tel que celui utilisé dans la première partie du TP. Afin d'arriver à cet objectif nous nous familiariserons avec le bus de communication CPU-FPGA et verrons les pièges.

2 Environnement de travail

L'ensemble des développements ainsi que la compilation se fait sur la machine hôte. Les binaires devront ensuite être déplacés dans un répertoire de l'ordinateur disponible depuis la carte par un montage réseau. Les manipulations, sur la plate-forme, se ferons dans un terminal, au travers d'une communication série.

2.1 Compilation des applications

Pour compiler une application il est nécesaire d'ajouter dans le *PATH*, le lien vers le compilateur, un script, à "sourcer" est disponible dans le répertoire *tp_fpga*

```
source apf27_enseignement.ggm
```

Dans le cadre des tests de génération de signaux périodiques, des scripts sont disponibles pour automatiser la compilation :

Pour les applications Linux, la commande suivante devra être tapée :

```
make
```

Pour les applications temps-réelles la commande sera :

```
make -f Makefile.xenomai
```

1. on prendra cependant soin de noter que ce 100 MHz est issu d'une multiplication par PLL et donc d'une stabilité médiocre – on préfèrera fournir une horloge externe de bonne qualité pour des mesures précises de temps – avec le problème de transfert des données entre les domaines cadencés par l'horloge externe et l'espace cadencé par l'horloge du CPU, nécessaire pour synchoniser les échanges sur les bus communs aux composants.

2.2 Obtenir un terminal sur la carte

L'ensemble des manipulation (flasher le FPGA et executer une application) se fait sur la carte. Pour obtenir une terminal il faut utiliser la commande :

```
minicom –D /dev/ttyS0
```

2.3 Accès aux binaires depuis la plate-forme

Au lieu de transférer les fichiers sur la carte APF27 nous allons utiliser un service réseau nommé **NFS** (*Network File System*). Grâce à celui-ci, il est possible de monter un répertoire de l'ordinateur sur la carte et ainsi d'accéder directement à son contenu de manière transparente (fig. 1).

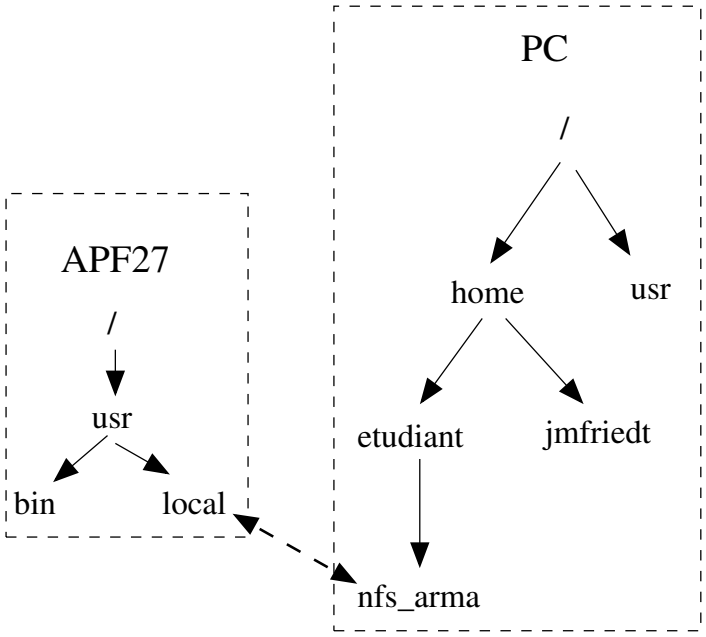


FIGURE 1 – Principe de l'utilisation de NFS : Le contenu du répertoire /home/etudiant/nfs_arma du PC est accessible directement depuis la carte APF27 au travers du répertoire /usr/local.

Pour ce faire, il faut, sur la carte, taper la commande suivante :

```
# mount 192.168.0.1:/home/etudiant/nfs_arma /usr/local
```

Avec :
— *192.168.0.1*, l'adresse correspondant à l'IP du PC (à adapter à chaque poste) ;
— */home/etudiant/nfs_arma* le répertoire du PC qui doit être monté ;
— */usr/local* l'endroit, dans l'arborescence du système de fichier de la carte, où il sera monté.
À partir de maintenant tous les binaires (.bit pour le FPGA et applications) devront être copiés dans le répertoire */home/etudiant/nfs_arma* du PC et l'ensemble du travail sur la carte devra être fait dans le répertoire */usr/local*

2.4 Flasher le FPGA

Dans le cadre des cartes APF27, le chargement du FPGA ne nécessite pas de composants externe : cette étape est réalisée à partir d'un terminal sur la carte et consiste à lancer la commande :

```
load_fpga nom_du_binaire.bit
```

3 Qualification Linux et Xenomai

L'ensemble des applications se trouve dans le répertoire */home/etudiant/tp_fpga/apps/imx27_gpio*

Afin d'évaluer le comportement des différents domaines dans le cas d'un système non chargé et chargé nous allons générer de manière logicielle un signal périodique. La période en sera de $1ms$. Il nous faudra donc changer l'état d'une broche (broche PC30 correspondant à la pin 3 du connecteur J9 de la carte APF27Dev) toutes les $500\mu s$.

Pour réaliser cette suite d'opération nous allons mettre en œuvre deux approches :

1. une mise en veille de la tâche pendant le temps fixé ;
2. l'utilisation d'un timer.

3.1 Génération d'un signal sur une mise en sommeil

```
1 int main(void)
  {
3     configuration_gpio;

5     while(1) {
          changement_etat_broche
7         mise_en_sommeil
      }
9 }
```

gpio_sleep.c : Principe de la génération d'un signal périodique dans le domaine Linux.

```
1
  void blink(void *arg)
3 {
      while(1) {
5         changement_etat_broche
          mise_en_sommeil
7     }

9 }

11 int main(void)
   {
13     configuration_gpio;
       creation_task
15     demarrage_task
       attente
17     destruction_task

19 }
```

xeno_gpio_sleep.c : Principe de la génération d'un signal périodique dans le domaine Xenomai.

3.1.1 Manipulation des broches du processeur depuis l'espace utilisateur

La configuration des broches se fait en deux étapes :
initialisation des couches de communication avec la fonction :

```
1 int imx27_gpio_init(void);
```

Puis configuration de la broche utilisée :

```
1 int imx27_gpio_set_cfgpin(unsigned int pin, unsigned int val);
```

avec :

— **pin** valant : IMX27_GPX(num) où X correspond au port (A, B, C, D, E) et **num** au numéro de la pin. Dans notre cas nous utiliserons la broche 30 du port C

— **val** valant *INPUT* ou *OUTPUT*

Le changement d'état d'une broche se fait au travers de la fonction :

```
int imx27_gpio_output(unsigned int pin, unsigned int val);
```

avec :

— **pin** qui correspond à la même référence que pour imx27_gpio_set_cfgpin() ;

— **val** qui peut prendre les valeurs *HIGH* ou *LOW* (ou plus simplement 1 ou 0).

Attention : Il est impératif de tester la valeur de retour des fonctions (retourne une valeur différente de 0 en cas de problèmes) pour s'assurer du bon fonctionnement, ie. :

```
if (function()) {
2   printf("erreur\n");
    return -1;
4 }
```

3.1.2 Mise en sommeil du process

Dans le cas d'une application Linux la fonction à utiliser est :

```
int usleep(useconds_t usec);
```

Où *usec* est un entier correspondant à une durée exprimée en *us*
Dans le cas de Xenomai la fonction sera :

```
int nanosleep(const struct timespec *rqtp, struct timespec *rmtp);
```

avec :

— rqtp une structure préalablement initialisée :

```
struct timespec tim = {0,TIMESLEEP};
```

et passée par pointeur (préfixée d'un '&'). TIMESLEEP correspond à un entier exprimé en nanosecond ;

— *rmtp* non utilisé et mis à **NULL**.

3.1.3 Création d'une tâche temps-réelle

Une application Xenomai est lancée depuis l'ordonnanceur temps-partagé, pour être en mesure de basculer sur l'ordonnanceur temps-réel il est nécessaire de créer une tâche qui réalisera les traitements.
La fonction :

```
int rt_task_spawn(RT_TASK *task, const char *name, int stksize, int prio, int mode,
    void(*)(void *cookie)entry, void * cookie);
```

Permet de créer la tâche et de la démarrer, les paramètres sont :

— *task* : un pointeur sur la tâche préalablement définie ;
— *name* : le nom de la tâche (ou NULL).
— *stksize* : la taille de la stack. Lui passer 0 laisse le système décider ;
— *prio* : la priorité affecté à la tâche, pour nos tests nous allons utiliser 99 (la priorité la plus haute) ;
— *mode* : **T_JOINABLE** pour laisser le thread principal attendre la terminaison de la tâche ;
— *entry* : un pointeur sur la fonction qui sera utilisée pour la tâche ;
— *cookie* : pour passer à cette fonction une information (NULL si non nécessaire).

Pour éviter que la fonction *main* ne s'arrête avant la terminaison de la tâche il est nécessaire d'utiliser la fonction :

```
int rt_task_join(RT_TASK *task);
```

dont le paramètre est la *RT_TASK* du premier paramètre de *rt_task_spawn()*

3.2 Génération d'un signal sur timer

```
1
void blink(int signum)
3 {
      changement_etat_broche
5 }

7 int main(void)
  {
9     configuration_gpio;

11    configuration_gestionnaire_event
      configuration_timer
13
      while(1) {}
15 }
```

gpio_sigalarm.c : Principe de la génération d'un signal périodique dans le domaine Linux.

```
1
void blink(void *arg)
3 {
      configuration_timer
5
      while(1) {
7         attente_timer
          changement_etat_broche
9     }
}
11
int main(void)
13 {
      configuration_gpio;
15    creation_task
      demarrage_task
17    attente
      destruction_task
19 }
```

xeno_gpio_timer.c : Principe de la génération d'un signal périodique dans le domaine Xenomai.

La partie configuration de la GPIO et la création d'une tâche sur l'ordonnanceur Xenomai ne change pas.

3.2.1 Configuration d'un timer sur ordonnanceur Linux

La gestion d'un timer sur événement (appel d'une fonction type gestionnaire d'interruption) se fait en plusieurs parties :
Enregistrement de la fonction comme devant être appelée sur le signal SIGVTALRM
Ceci se fait au travers de la structure

```
struct sigaction sa;
```

Qui doit, dans un premier temps, être initialisée à '0' grâce à

```
memset(void *ptr, int c, size_t n);
```

où :
- *ptr* est un pointeur sur la structure (utilisation de '&') ;
- *c* correspond au caractère à utiliser ('0') ;
- *n* le nombre d'octets à écrire (**sizeof(sa)**).

La seconde étape est de fournir à cette structure un pointeur sur la fonction qui sera appelée lors du déclenchement de l'événement :

```
sa.sa_handler = &test;
```

Et pour finir il faut enregistrer notre gestionnaire pour répondre au bon événement grâce à la fonction :

```
int sigaction(int signum, const struct sigaction *act,
                    struct sigaction *oldact);
```

où :
- signum correspond à l'événement (SIGVTALRM) ;
- act : la structure précédemment remplie ;
- oldact : non utilisée (NULL), permet de sauvegarder la configuration courante.

configuration de la génération d'un signal à intervalle régulier sur ordonnanceur Linux
Cette étape repose sur le remplissage d'une structure :

```
struct itimerval timer;
```

4 champs plus précisément doivent être remplis :
- it_value.tv_sec : déclenchement initial après 'n' sec (donc 0)
- it_value.tv_usec : déclenchement initial après 'n' usec (donc 500)
- it_interval.tv_sec : déclenchement périodique après 'n' sec (donc 0)
- it_interval.tv_usec : déclenchement périodique après 'n' usec (donc 500)

Et pour finir, la fonction :

```
int setitimer(int which, const struct itimerval *new_value,
                    struct itimerval *old_value);
```

- which définie le compteur du timer et le signal qui sera émis (ITIMER_VIRTUAL) ;
- new_value : la structure préalablement remplie ;
- old_value : non utilisée (donc NULL) pour mémoriser la configuration courante.

3.2.2 Configuration d'un timer sur ordonnanceur Xenomai

L'utilisation d'un timer pour réaliser un traitement périodique se fait en deux temps :
la configuration :

```
int rt_task_set_periodic(RT_TASK *task, RTIME idate, RTIME period);
```

avec :
- *task* un pointeur sur une tâche créée précédemment. Si NULL alors la tâche courante deviens périodique ;
- *idate* date de départ. si **TM_NOW** alors aucun délai avant le démarrage ;
- *period* durée de la période en nanoseconde.

Ensuite dans la boucle la fonction :

```
int rt_task_wait_period(unsigned long *overruns_r);
```

où overruns_r nombre de dépassement (ou NULL si non utilisé). Cette fonction a pour rôle de suspendre la tâche jusqu'à expiration du timer.

3.3 Utilisation

3.3.1 Compteur de période

Le binaire à destination du FPGA se trouve dans le répertoire */home/etudiant/tp_fpga* et se nomme *top_exercice_counter.bit*. Il est également nécessaire de compiler, et d'installer, l'application */home/etudiant/tp_fpga/apps/app_exercice_counter*

Les applications préalablement copiées dans */home/etudiant/nfs_arma* sont disponibles depuis la plate-forme dans le répertoire */usr/local*

En tout premier lieu il est nécessaire de charger le binaire dans le FPGA par la commande :

```
load_fpga top_exercice_counter.bit
```

la phase de comptage et de stockage de résultat se fait en tapant la commande :

```
./app_exercice_counter
```

3.3.2 Test des diverses solutions

Ne disposant que d'un seul terminal sur la carte les applications doivent être lancées en tâche de fond (utilisation d'un '&' après le nom de l'applis.

Pour arrêter l'application il est nécessaire de la *tuer* :

```
# ps aux
PID   USER     COMMAND
  625 root      ./gpio_sleep
#kill 625
```

Une fois l'application chargée, une série d'acquisition sera faite en exécutant la commande ./app_exercice_counter. Quand celle-ci rends la main, un fichier **counter.dat** est crée dans le répertoire courant. Ce fichier doit être renommé pour être représentatif du test.

Pour charger le système, un script, nommé *charge.sh* est disponible dans */home/etudiant/tp_fpga/apps/imx27_gpio*. Pour le lancer :

```
./charge.sh &
```

Pour l'arrêter il faut supprimer le fichier *attente.nop* (commande **rm**).

3.3.3 Exploitation des résultats

L'exploitation se fera grâce à *octave* sur la machine hôte. Les commandes à utiliser sont les suivantes :

```
% chargement du fichier
a = load("fichier.dat");
% base de temps 10ns => /100 pour us
a=a./100;
% affichage histogramme
hist(a)
```

Pour avoir un meilleur détail sur l'histogramme il est également possible d'utiliser :

```
figure;[xx,nn]=hist(a,[min(a):10:max(a)]);bar(nn,log(xx+1))
```

Attention : Selon la valeur des limites, et du pas, cette commande peut être très lente.

4 Communication CPU-FPGA

Nous avons pu voir, dans la première partie du TP, les caractéristiques temporelles de Linux et de Xenomai, les mesures ont put être faites grâce à un design dédié dans le FPGA (composant temps-réel par nature). Dans la seconde partie de ce TP, nous allons nous focaliser sur la partie FPGA. Avec pour but final de réaliser le compteur utilisé pour qualifier les diverses solutions.

Pour atteindre ces objectifs, nous allons passer par les étapes de développement
intermédiaires que sont :

1. communication processeur-FPGA (noyau Linux, bus de communication dans le FPGA),
2. implémentation d'un compteur dans le FPGA et transfert de la valeur au CPU,
3. utilisation du FPGA pour mesurer le *jitter* d'un signal produit par le processeur générique, avec et sans Xenomai.

5 Environnements matériel et logiciel

Comme présenté plus tôt, le matériel utilisé, une carte Armadeus Systems APF27, dispose d'un processeur générique couplé à un FPGA. Un bus de communication rapide connecte les deux composants (fig 2).

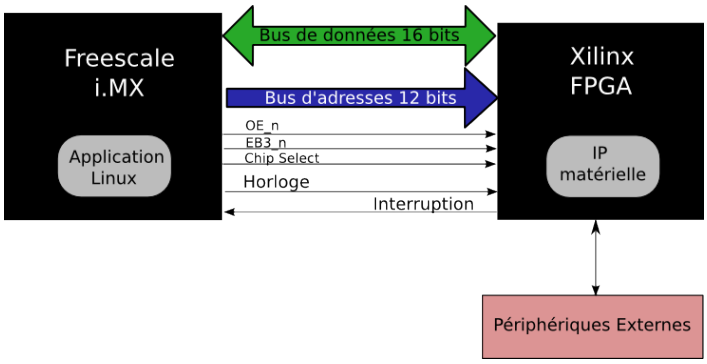


FIGURE 2 – Environnement matériel utilisé : carte APF27.

Notre premier développement logiciel concerne le protocole de communication entre blocs de traitement dans le FPGA : chaque bloc est codé en VHDL, et le protocole de communication entre blocs ainsi que entre CPU et FPGA doit être implémenté dans ce langage.

Afin d'autoriser plusieurs composants HDL à être accessibles depuis le CPU et de simplifier la partie communication, côté FPGA, *Armadeus* propose une implémentation du bus *Wishbone*. Avec ce protocole, le CPU est toujours maître et les composants dans le FPGA esclaves. Une partie importante des transactions (décodage d'adresse en particulier) est réalisé par le bloc **Intercon** (fig. 3), automatiquement généré par le logiciel *Peripheral On Demand*[2] qui est un outil d'assemblage de blocs HDL (non traité dans ce TP). Ainsi la plage d'adresses correspondant à l'espace mémoire entre le CPU et le FPGA peut être découpée en plusieurs zones correspondant à chaque composant (figure 4).

5.1 Génération du binaire

La génération du binaire à partir du design VHDL est réalisé par l'application *ise*.

Pour lancer cette application il est d'abord nécessaire de compléter ses variables d'environnement par :

```
source /opt/Xilinx/14.7/ISE_DS/settings32.sh
```

De taper la commande

```
ise&
```

Puis d'ouvrir le projet et finalement d'appuyer sur *generate bitstream*

6 Découverte de la communication FPGA-CPU : addition

Le projet VHDL se trouve dans le répertoire /home/etudiant/tp_fpga/design/exercice_addition et le fichier à ouvrir dans ise dans le sous-répertoire objs/exercice_addition.xise

Ce premier exercice est l'occasion de se familiariser avec la communication au travers du bus Wishbone. Pour ce faire nous allons réaliser une IP (Intellectual Property), ou composant, qui réalise une addition.

2. http://github.com/martoni/periphondemand

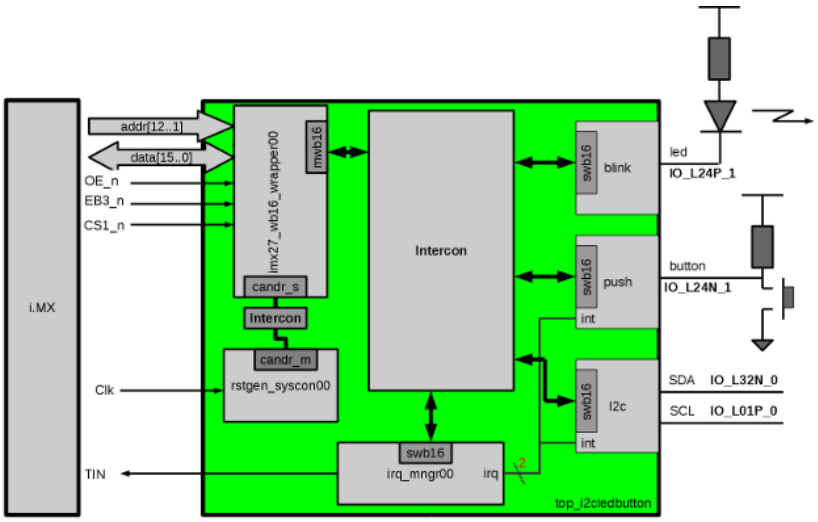


FIGURE 3 – Exemple de design FPGA

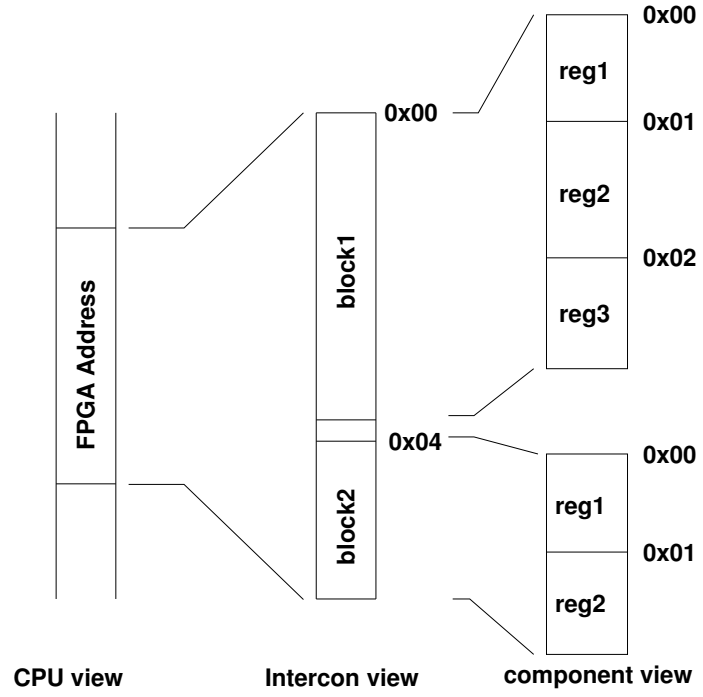


FIGURE 4 – Principe d'abstraction des adresses des composants dans le FPGA. La plage d'adresse dédiée à la communication entre le CPU et le FPGA est découpée en plusieurs sous-adresses. L'intercon réalise le décodage et adresse le composant visé en lui fournissant le numéro du registre correspondant à la transaction.

6.1 Principe de base de la communication Wishbone

Compte tenu de la structuration imposée par *Peripheral On Demand*, une IP, dans cet environnement, ne doit prendre en compte qu'un sous ensemble de la complexité du protocole Wishbone et de ses signaux (signaux **wbs_xx** sur la figure 5).

L'IP reçoit trois signaux de contrôle :

```
entity enseignement_addition is
  generic (
    id          : natural := 1;
    wb_size     : natural := 16); -- Data port size for wishbone
  port (
    -- Syscon signals
    reset       : in std_logic;
    clk         : in std_logic;
    -- Wishbone signals
    wbs_add         : in std_logic_vector(2 downto 0);
    wbs_writedata : in std_logic_vector(wb_size-1 downto 0);
    wbs_readdata_o  : out std_logic_vector(wb_size-1 downto 0);
    wbs_strobe      : in std_logic;
    wbs_cycle       : in std_logic;
    wbs_write       : in std_logic;
    wbs_ack         : out std_logic);
end entity enseignement_addition;
```

FIGURE 5 – Exemple d'entité d'une IP disposant d'une communication wishbone.

— **wbs_strobe** et **wbs_cycle** qui indiquent la validité d'une transaction ;
— **wbs_write** qui, à l'état haut, indique une requête d'écriture et à l'état bas une lecture.

Chaque requête doit être acquitée en passant à l'état haut le signal **wbs_ack**.

À ces signaux de contrôle s'ajoutent :

— un bus d'adresse (**wbs_add**) qui fournit les adresses relatives au composant. L'adresse absolue, du point de vue CPU, est gérée par celui-ci, tandis que l'adresse relative de chaque composant est gérée par le bloc *intercon*. Le composant ne reçoit donc que les addresses *relatives* des registres qui lui sont propres (tel qu'illustré sur la figure 4). La taille du bus dépends du nombre de registres du bloc et est donc fixée par le développeur ;
— un bus de données pour l'écriture (**wbs_writedata**) (en mode **in**) ;
— un bus de données pour la lecture (**wbs_readdata_o**)(en mode **out**).

6.2 Implémentation du bloc d'addition

Compte tenu de la relative simplicité de notre composant, celui-ci dispose de 4 registres :

```
constant REG_ID     : std_logic_vector(2 downto 0) := "000";
constant REG_OP1    : std_logic_vector(2 downto 0) := "001";
constant REG_OP2    : std_logic_vector(2 downto 0) := "010";
constant REG_RESULT : std_logic_vector(2 downto 0) := "011";
```

— un registre d'identifiant (RO). Celui-ci est obligatoire. Il se trouve à l'adresse 0x00 ;
— deux registres pour fournir les opérandes (WO), que nous placerons aux adresses 0x01 et 0x02 ;
— un dernier registre pour récupérer le résultat de l'opération à l'adresse 0x03 (RO).

Par habitude, et pour rendre le code plus lisible, les requêtes de lecture et d'écriture (toujours du point de vue du processeur) sont séparées en deux process.

6.3 Gestion de la lecture

Un premier process est utilisé pour les phases de lecture (du point de vue du processeur). Les deux registres disposant d'un accès en lecture sont traités (listing ci-dessous) :

— le registre **REG_ID** pour un identifiant unique (0x00) ;
— le registre **REG_RESULT** pour transférer le résulat de l'opération (0x03).

```
read_bloc : process(clk, reset)
begin
  if reset = '1' then
    wbs_readdata_s <= (others => '0');
  elsif rising_edge(clk) then
    wbs_readdata_s <= wbs_readdata_s;
    if ((wbs_cycle and wbs_strobe ) = '1' and wbs_write = '0' ) then
      case wbs_add is
      when REG_ID =>
```

```
10          wbs_readdata_s <= std_logic_vector(to_unsigned(id, wb_size));
        when REG_RESULT =>
12          wbs_readdata_s <= result_s;
        when others =>
14          wbs_readdata_s <= (others => '0');
        end case;
16      end if;
    end if;
18  end process read_bloc;
```

Dans ce process nous voyons que les signaux **wbs_cycle** et **wbs_strobe** sont évalués et nous nous intéressont à leur état haut (une transaction est en cours). Nous évaluons également le signal **wbs_write** qui doit se trouver à l'état bas pour signaler que la requête est une lecture. Si cette condition est validée alors nous évaluons le signal **wbs_add** dans un **case** pour connaître le registre demandé, selon celui-ci le signal FPGA vers CPU **wbs_readdata_s** est affecté avec le contenu du signal correspondant. Si l'adresse n'est pas un registre valide alors nous fournissons une valeur quelconque.

6.4 Gestion de l'écriture

Dans le cas du process en charge de la gestion de l'écriture seuls les deux registres utilisés pour la configuration des opérandes sont gérés, les autres étant en lecture seule.

```
  write_bloc : process(clk, reset)   -- write DEPUIS l'iMx
2 begin
    if reset = '1' then
4     op1_s <= (others => '0');
      op2_s <= (others => '0');
6   elsif rising_edge(clk) then
      op1_s <= op1_s;
8     op2_s <= op2_s;
     if ((wbs_cycle and wbs_strobe ) = '1' and wbs_write = '1' ) then
10      case wbs_add is
        when REG_OP1 =>
12        op1_s <= wbs_writedata;
        when REG_OP2 =>
14        op2_s <= wbs_writedata;
        when others =>
16      end case;
        end if;
18  end if;
  end process write_bloc;
```

Comme pour la lecture, nous évaluons les deux signaux **wbs_strobe** et **wbs_cycle**, ainsi que **wbs_write** mais cette fois-ci en testant qu'il est à l'état haut. Également comme dans le cas de la lecture, nous évaluons la valeur de **wbs_add** pour connaître quel registre est accéder. Selon le cas nous affectons la valeur contenu dans **wbs_writedata** à **op1_s** ou **op2_s**. Dans le cas où l'adresse ne correspond à aucun signal valide nous ne faisons rien.

6.5 Communication avec le FPGA

Pour ce premier exercice nous allons faire usage de l'utilitaire **fpgaregs** qui permet d'accéder directement à la zone mémoire partagée, en lecteur et en écriture. Il s'utilise ainsi :

```
1 fpgaregs w 0x10 --> read @ 0x10
  fpgaregs w 0x10 0x1234 --> writes 0x1234 @ 0x10
```

le *w* signifie word, soit des écriture en 16 bits.

L'adresse de base de notre IP est 0x08.

Deux points importants à prendre en compte :

1. **fpgaregs** gére automatiquement l'adresse physique ainsi les adresses à lui passer sont relatives ;
2. la communication se fait exclusivement en 16 bits (soit 2 octets), donc passer d'un registre au suivant se fait en incrémentant de 2. Ou d'une autre mesure accéder à un registre nécessite de réaliser un décallage de 1 vers la gauche (multiplication par 2) pour aligner les accès sur des adresses 16 bits.

Lecture de l'ID du bloc (addresse 0x08 + 0x00) :

```
# fpgaregs w 0x08
Read 0x0002 at 0x00000008
```

Écriture de la valeur 2 dans le registre **REG_OP1** (0x0A = 0x08 + 0x01 << 1)

```
# fpgaregs w 0x0A 2
Write 0x0002 at 0x0000000A
```

Écriture de la valeur 3 dans le registre **REG_OP2** (0x0C = 0x08 + 0x02 << 1)

```
# fpgaregs w 0x0C 3
Write 0x0003 at 0x0000000C
```

Lecture du résultat (5) depuis le registre **REG_RESULT** (0x0E = 0x08 + 0x03 << 1)

```
# fpgaregs w 0x0E
Read 0x0005 at 0x0000000E
```

6.6 Exercice

1. complétez les process de lecture et d'écriture ;
2. les registres **REG_OP1** et **REG_OP2** sont en écriture seule. En se basant sur les registres **REG_ID** et **REG_RESULT**, ajoutez le code nécessaire pour relire, depuis le processeur, leurs valeurs ;
3. Le bloc ne fait, en l'état, que l'addition : Ajoutez un registre pour configurer le bloc pour, soit réaliser une addition, soit une soustraction. Il est également nécessaire de modifier l'opération asynchrone pour gérer les deux traitements possibles.

Aide : une opération ou affectation conditionnelle se fait, en asynchrone, grâce au test :

```
signal_destination <= operation_ou_valeur_si_condition_vrai when condition
  else operation_ou_valeur_si_condition_fausse;
```

7 Lecture d'une RAM depuis le CPU

Le projet VHDL se trouve dans le répertoire /home/etudiant/tp_fpga/design/exercice_ram et le fichier à ouvrir dans ise dans le sous-répertoire objs/exercice_ram.xise

Un point important concernant la communication, et qui n'a pas été évoqué dans l'exercice précédent, concerne la durée de chaque étape de communication. En effet, les signaux de contrôle peuvent rester à l'état haut pendant plus d'un cycle d'horloge, comportement qui n'est pas souhaitable si notre IP incrémente un compteur à chaque requête. En effet, dans ce cas le compteur sera réellement incrémenté plusieurs fois. Si ce compteur sert comme indice dans l'accès à une RAM, des données seront perdues, ou des données seront duppliquées dans le cas d'une écriture.

Pour éviter ce cas de figure, il nous est nécessaire d'ajouter un nouveau process qui détectera uniquement le front montant du signal **wbs_strobe** : il nous sera ainsi possible de rendre notre traitement atomique (garantir que chaque traitements ne dure qu'une période de l'horloge cadençant les transactions sur le bus de communication) et ainsi éviter que l'adresse de la RAM dans le cas présent ne soit incrémenté plus qu'une fois par transaction.

Notre second exercice sera donc de réaliser un bloc qui :

1. suite à la reception d'un ordre de start du processeur, démarre une pseudo acquisition. Celle-ci consiste uniquement à démarrer un compteur qui ira stocker dans une RAM la valeur de l'adresse ;
2. met à l'état haut un signal d'état pour signifier au processeur si l'acquisition est en cours ou fini. Le processeur pourra lire la valeur de ce signal au travers d'un registre de status et le sondera par *polling* ;
3. transmettre au processeur le contenu du tableau en incrémentant à chaque nouvelle requête de lecture la valeur d'un compteur d'adresse à la manière d'une *FIFO*.

Ainsi nous avons besoin de 4 registres :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ID															

registre 0x00 : dédié à fournir un identifiant unique. Lecture seule.

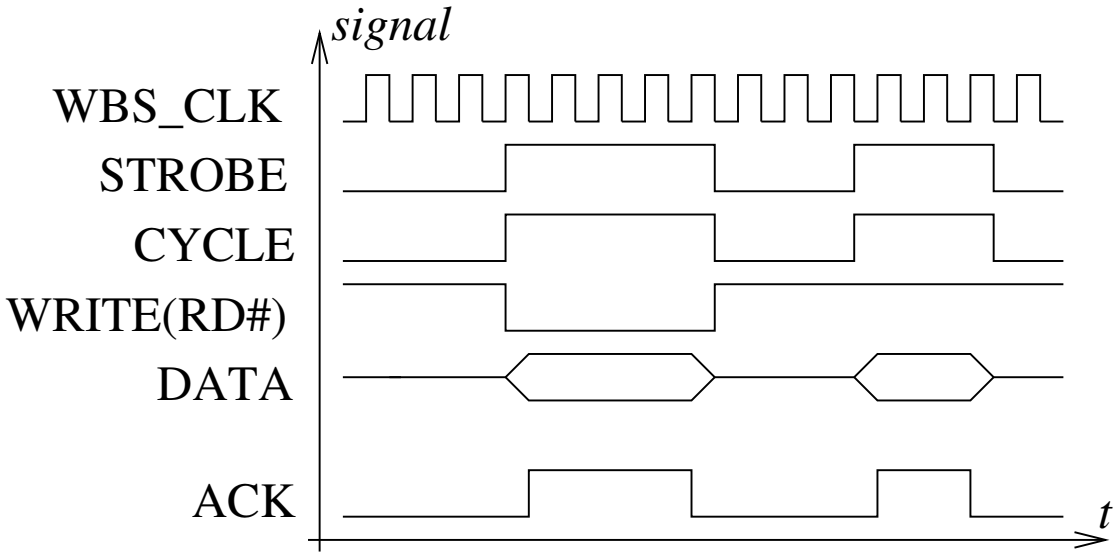


FIGURE 6 – wbs_clk, strobe, cycle, write et data viennent du processeur. ack est géré

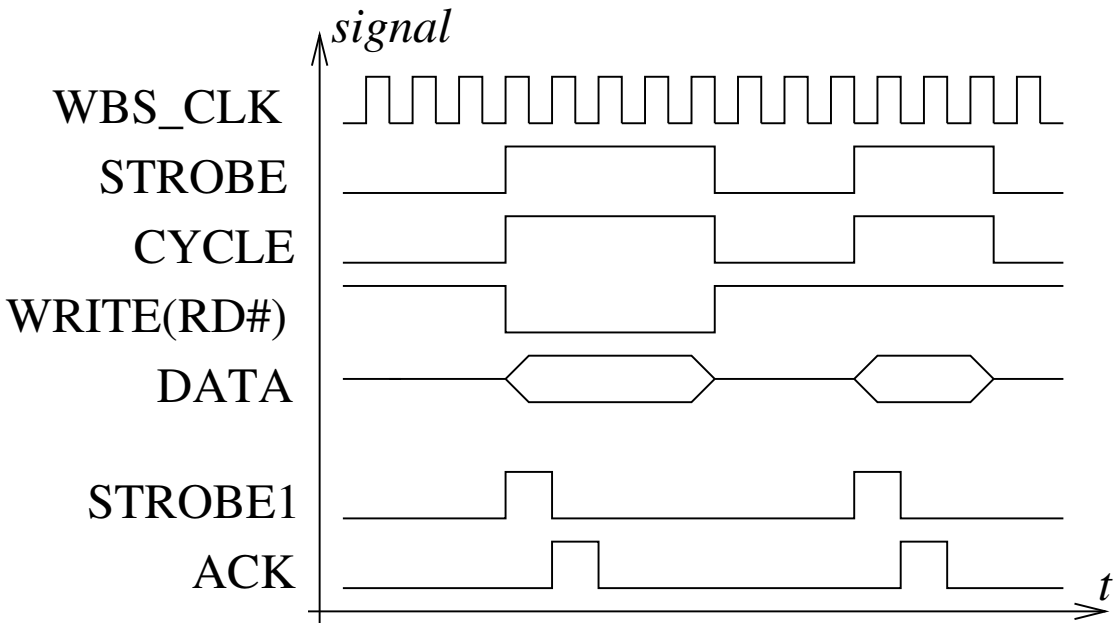


FIGURE 7 – wbs_clk, strobe, cycle, write et data viennent du processeur. ack est géré pendant un seul cycle d'horloge

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X															busy

registre 0x01 : lecture seule. Registre de status. **busy** reste à 1 tant qu'une acquisition est en cours. Est à zéro sinon.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X															start

registre 0x02 : lecture-écriture. Fournit l'ordre depuis le processeur du démarrage de l'acquisition.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
data															

registre 0x03 : lecture seule. Utilisé pour récupérer les valeurs succéssives contenues dans la RAM.

Le language VHDL étant relativement complexe, et surtout verbeux, et pour rendre le code plus lisible, ce bloc est découpé en trois fichiers :

- un fichier encapsulant la RAM ;
- un fichier pour la gestion de la communication ;
- un fichier **"top"** qui contient, d'une part le process de simulation d'une acquisition, mais également les instanciations de la RAM et du bloc de gestion de la communication.

7.1 Pseudo acquisition : synthesis/enseignement_ram/enseignement_ram.vhd

Le process en charge de simuler l'acquisition d'un flux de données comporte deux états (représentés par le signal **busy**) :

- lorsque ce signal est au niveau bas (l.13-17), le process est inactif et attends l'ordre d'acquisition du processeur (signal **start_acquisition_s**). Quand ce dernier signal passe, pendant un cycle d'horloge, à l'état haut (l.14-17), le process met son compteur d'adresse à zéro et passe le signal **busy** à l'état haut ;
- lorsque le signal est au niveau haut (l.18-27), le process incrémente un compteur à chaque cycle d'horloge et envoie un ordre d'écriture à la RAM. Lorsque le compteur arrive à la valeur $2^{10} - 1$ le signal **busy** repasse à l'état bas et le process se met à au nouveau dans l'état d'attente.

Comme le signal **busy** est à l'état haut pendant tout le temps de l'acquisition, ce signal est propagé vers la partie gestion de la communication pour donner au processeur l'état du traitement.

```
cpt_storage_proc: process(clk, reset)
begin
  if reset = '1' then
    busy_s <= '0';
    cpt_addr_s <= (others => '0');
    cpt_addr_next_s <= 0;
    cpt_en_s <= '0';
  elsif rising_edge(clk) then
    cpt_en_s <= '0';
    cpt_addr_s <= cpt_addr_s;
    cpt_addr_next_s <= cpt_addr_next_s;
    busy_s <= busy_s;
    if busy_s = '0' then -- state idle
      if start_acquisition_s = '1' then
        busy_s <= '1';
        cpt_addr_next_s <= 0;
      end if;
    else -- state storage
      cpt_addr_s <= std_logic_vector(to_unsigned(cpt_addr_next_s, 10));
      cpt_data_s <= std_logic_vector(to_unsigned(cpt_addr_next_s+1, 16));
      cpt_en_s <= '1';
      if cpt_addr_next_s = 1023 then
        busy_s <= '0';
      else
        cpt_addr_next_s <= cpt_addr_next_s+1;
      end if;
    end if;
  end if;
end process;
```

7.2 Communication Wishbone : synthesis/enseignement_ram/ens_ram_wb.vhd

Dans un premier temps, pour conserver les acquis du premier exercice, et voir l'impact de l'absence de gestion des signaux de contrôles restant à l'état pendant plusieurs cycles d'horloge, nous n'allons pas ajouter de mécanisme de gestion. Ceci sera fait dans un second temps.

La partie écriture de notre bloc de communication est très simple :

```
write_bloc : process(clk, reset)   -- write DEPUIS l'iMx
begin
   if reset = '1' then
    start_acquisition_o <= '0';
   elsif rising_edge(clk) then
    start_acquisition_o <= '0';
    if ((wbs_strobe and wbs_write and wbs_cycle) = '1' ) then
```

```
      case wbs_add is
      when REG_START =>
        start_acquisition_o <= wbs_writedata(0);
      when others =>
      end case;
      end if;
   end if;
end process write_bloc;
```

Un seul registre est géré : **REG_START** qui copie le bit de poids faible du registre d'écriture dans le signal **start_acquisition_o**. Ce dernier est propagé au process d'acquisition présenté dans la section précédente pour lancer une phase d'acquisition. Il est à remarquer toutefois que la valeur de ce signal n'est pas conservé : au prochain cycle d'horloge, dès lors que les conditions d'écriture ne sont plus validées, le signal est remis au niveau bas.

le process de lecture, par contre, présente plus de complexité :

```
read_bloc : process(clk, reset)
begin
  if reset = '1' then
    wbs_readdata_s <= (others => '0');
    data_addr_s <= (others => '0');
  elsif rising_edge(clk) then
    data_addr_s <= data_addr_s;
    wbs_readdata_s <= wbs_readdata_s;
    if (wbs_strobe = '1' and wbs_write = '0'  and wbs_cycle = '1' ) then
      case wbs_add is
      when REG_ID =>
         wbs_readdata_s <= std_logic_vector(to_unsigned(id,wb_size));
      when REG_STATUS =>
        wbs_readdata_s <= (15 downto 1 => '0')&busy_i;
      when REG_DATA =>
          wbs_readdata_s <= data_val_i;
          data_addr_s <= std_logic_vector(unsigned(data_addr_s)+1);
      when others =>
        wbs_readdata_s <= x"AA55";  -- lecture dans reg inconnu
      end case;
    end if;
  end if;
end process read_bloc;
```

En plus de gérer l'id il doit gérer :

- les requêtes pour connaître l'état du bloc (occupé ou inactif) (l.13-14). Ce traitement consiste à concaténer un ensemble de bit à l'état bas ((15 downto 1 => '0')) et comme bit de poids faible le signal **busy_i** correspondant au signal du process d'acquisition ;
- les requêtes pour l'obtention des données succéssives contenues dans la RAM (l.15-17). Ce traitement consiste à copier la valeur portée par le bus de donnée de la RAM (signal **data_val_i**) dans le bus de données FPGA vers CPU et d'avancer l'index de la RAM pour la requête de lecture suivante. Il est à noter que lorsque le signal **data_addr_s** atteindra 0x3ff son incrémentation remettra ce signal à 0x000.

7.3 Communication depuis le processeur

L'outil *fpgaregs* n'est pas adapté pour gérer un nombre important d'accès. Par ailleur, cet utilitaire permet d'écrire dans le FPGA ou d'afficher la valeur d'un registre mais ne retourne pas celle-ci à la console.

C'est pourquoi nous allons utiliser une application en espace utilisateur pour communiquer avec le FPGA.

Du point de vue Linux, nous devons convertir l'adresse virtuelle exploitée par le processeur au travers de la MMU, vers une adresse physique qui sera transmise sur le bus d'adresse. La fonction sous GNU/Linux chargée de faire cette conversion est la fonction `mmap()`. Alternativement, depuis l'espace utilisateur nous accédons à l'espace des adresses physiques au travers du pseudo-fichier `dev/mem`.

Ainsi la première étape pour accéder au FPGA consiste à ouvrir le pseudo fichier */dev/mem* en lecture et écriture :

```
int fd = open("/dev/mem", O_RDWR|O_SYNC);
if (fd < 0) {
  printf("can't open file /dev/mem\n");
  return -1;
}
```

À partir du descripteur de fichier **fd** nous pouvons ensuite demander, à l'aide de la fonction *mmap*, un pointeur sur une zone mémoire liée à l'adresse physique à laquelle nous souhaitons accéder :

```
void *ptr_fpga = mmap(0, 8192, PROT_READ|PROT_WRITE, MAP_SHARED,
        fd, 0xD6000000);
if (ptr_fpga == MAP_FAILED) {
  printf("mmap failed\n");
  return -2;
}
```

et finalement réaliser des lectures de la manière suivante :

```
unsigned int pos = fpga_offset+ registre;
unsigned short content = *(unsigned short*)(ptr_fpga+((unsigned short)(pos)));
```

et pour l'écriture :

```
unsigned int pos = fpga_offset + registre;
*(unsigned short*)(ptr_fpga + pos) = (unsigned short)value;
```

Ce qui nous donneras le code suivant :

```
int fd = open("/dev/mem", O_RDWR|O_SYNC);
if (fd < 0)
  return EXIT_FAILURE;
ptr_fpga = mmap(0, 8192, PROT_READ|PROT_WRITE, MAP_SHARED,
            fd, FPGA_BASE_ADDR);
if (ptr_fpga == MAP_FAILED)
  return EXIT_FAILURE;

pos = FPGA_OFFSET + REG_ID;
value = *(unsigned short*)(ptr_fpga+((unsigned short)pos));
if (value != 2)
  return EXIT_FAILURE;

/* start acquistion */
pos = FPGA_OFFSET + REG_START;
*(unsigned short*)(ptr_fpga+pos) = 0x01;

/* wait until busy low */
pos = FPGA_OFFSET + REG_STATUS;
do {
  value = *(unsigned short*)(ptr_fpga+((unsigned short)pos));
} while((value & 0x01) == 0x01);

/* receive data */
pos = FPGA_OFFSET + REG_DATA;
for (i=0; i<1024; i++) {
  value = *(unsigned short*)(ptr_fpga+((unsigned short)pos));
  printf("%hu\n", value);
}
```

- en premier lieu, il ouvre le pseudo fichier */dev/mem* (l.1-3), puis utilise *mmap* (l.4-7) pour obtenir un pointeur sur la zone mémoire correspondant au bloc dans le FPGA ;
- ensuite il lit le registre contenant l'identifiant et s'assure qu'il correspond à la valeur attendue (l.9-12) ;
- envoie l'ordre de démarrage d'une acquisition (l.15-16) et attends la fin de celle-ci(l.20-22) ;
- pour finir récupére séquentiellement l'ensemble des valeurs contenues dans la RAM (l.25-29) et les affiches dans le terminal.

Pour compiler ce programme il faut en premier lieu, sur le PC, taper la commande

```
source apf27_enseignement.ggm
```

Puis taper make dans le répertoire contenant les sources et finalement copier le binaire **app_exercice_ram** dans le répertoire */home/etudiant/nfs_arma* à l'aide de la commande **cp**.

Pour executer le programme et récupérer les valeurs dans un fichier. Sur la carte, dans le répertoire */usr/local*, il faut taper la commande ./app_exercice_ram > test1.dat (le caractére '>' est utilisé pour rediriger l'affichage du terminal vers le fichier test1.dat.

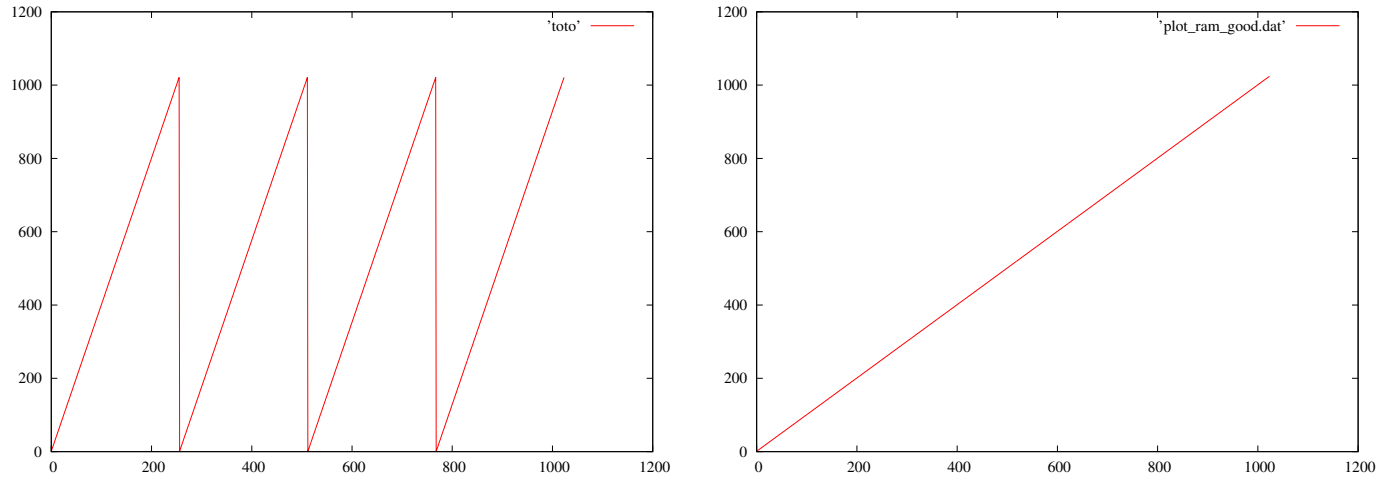


FIGURE 8 – Résultat de la récupération des données stockées dans la RAM. À droite sans gestion des durées des signaux de contrôle, droite avec la gestion

Complétez le code du design exercice_ram, puis lancez une acquisition. À l'aide d'octave affichez la courbe qui devrait correspondre à celle de la figure 8 à gauche.

```
a = load("test1.dat");
plot(a);
```

Au lieu d'avoir une ligne continu allant de 0 jusqu'à 1024 nous constatons une série de lignes. Ceci s'explique par la non gestion des durées de chaque transactions.

7.4 Ajout d'une détection du front montant de wbs_strobe

La première partie de ce traitement consiste à comparer l'état courant de **wbs_strobe** avec son état passé (ie. au précédent front montant de l'horloge). Ceci passe donc par la mémorisation de son état à chaque cycle de d'horloge, ainsi qu'à une comparaison :

```
-- comparaison
strobe_rise <= '1' when signal_old_s = '0' and wbs_strobe = '1' else '0';
-- memorisation
detection_front: process(clk, reset)
   begin
    if reset = '1' then
       signal_old_s <= '0';
     elsif rising_edge(clk) then
      signal_old_s <= wbs_strobe;
    end if;
end process detection_front;
```

La partie détection à proprement parler est réalisée de manière asynchrone pour ne pas retarder le déclenchement des traitements dans les process de lecture et d'écriture, ce qui aurait pour effet, dans le cas de la lecture, de transférer au processeur l'ancien état de **wbs_readdata_s** et non l'état courant.

La seconde partie du traitement consiste à modifier les deux process pour remplacer le signal **wbs_strobe** par **strobe_rise**

Pour lécriture :

```
if ((wbs_strobe and wbs_write and wbs_cycle) = '1' ) then
```

par :

```
if ((strobe_rise and wbs_write and wbs_cycle) = '1' ) then
```

Et pour la lecture :

```
if (wbs_strobe = '1' and wbs_write = '0'  and wbs_cycle = '1' ) then
```

par :

```
if (strobe_rise = '1' and wbs_write = '0'  and wbs_cycle = '1' ) then
```

Ajoutez le mécanisme de gestion et modifiez les process de lecture et d'écriture pour remplacer le signal **wbs_strobe** par **strobe_rise** et relancer une acquisition. Vous devez trouver une courbe équivalente à celle de droite sur la figure 8. En effet nous ne réagisons que pendant un seul cycle d'horloge et l'index de la RAM n'est plus incrémenté qu'une seule fois par transaction.

8 Réalisation du composant de mesure-stockage

Les deux premiers exercices ont permis, d'une part d'acquérir les bases de la communication entre le processeur et le FPGA dans le cadre des cartes APF27, et d'autre part de mettre en évidence certains pièges relatifs aux signaux de contrôles. Il est maintenant possible de réaliser le bloc compteur utilisé pour évaluer les latences d'une application en espace utilisateur GNU/Linux ou dans le domaine temps-réel de Xenomai.

Cet exercice a pour but de disposer d'un bloc capable de mesurer la durée entre deux fronts montants consécutifs, avec une profondeur de comptage capable de mesurer des durées des plusieurs secondes.

8.1 Implémentation du compteur de durée de périodes

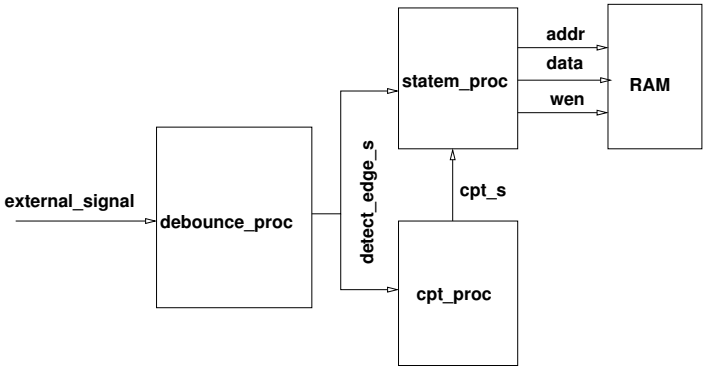


FIGURE 9 – Schéma global du traitement à implémenter

La partie compteur de ce composant est découpé en trois process (Fig. 9) :

1. un process *debounce_proc*. Il a pour rôle de "nettoyer" le signal à étudier. En effet ce signal est issu d'un système externe, il est donc possible que des **"glitchs"** puissent apparaître et fausser les résultats. Ce process pilote également un signal qui, s'il est à l'état haut, avertit les autres process de la détection d'un front montant.
2. un process *cpt_proc* qui compte le nombre de périodes de l'horloge du FPGA entre deux fronts montants du signal à étudier. Sa valeur est remise à zéro sur la détection du front et incrémentée sinon ;
3. un process *statem_proc*, proche du process de pseudo-acquisition de l'exercice 2, comporte une machine à états qui est par défaut en attente d'un ordre d'acquisition de la part du processeur. Lorsque que cet ordre est reçut, le process réalise n acquisition consécutives avant de se remettre dans l'état initial.

8.2 debounce_proc

Le principe de ce process est d'évaluer une série d'états passés du signal à étudier pour s'assurer que le signal est resté stable pendant cette durée.

Ce mécanisme repose sur un registre à décalage, mis à jour à chaque front montant de l'horloge de cadencement, qui maintient l'état du signal sur plusieurs cycles d'horloges. Ainsi si le registre est composé de '1' l'état est un état haut stable, s'il est composé complètement de '0' c'est un état bas stable.

Pour déterminer le changement d'état (transition bas-haut ou haut-bas), un signal est utilisé. Ainsi, si l'état déterminé par le registre est haut et que le signal vaut '0' il y a eut un front montant et réciproquement. Cette détection est utilisée pour propager l'information de front montant pour les traitements réalisés par les autres process.

8.3 cpt_proc

Le principe de ce process est d'évaluer, à chaque front montant de l'horloge, l'état du signal de détection du front montant :

— s'il est à l'état haut, le compteur est remis à 0 (utilisation de `(others => '0')`) ;

— s'il est à l'état base le signal est incrémenté par `cpt_s <= std_logic_vector(unsigned(cpt_s) + 1);`

Afin d'être capable de compter plusieurs secondes, le compteur est codé sur 32 bits.

8.4 statem_proc

Globalement ce process est proche de la pseudo-acqusition de l'exercice précédent.

Ce process comporte trois états :

1. un état IDLE : le process est en attente de l'état haut du signal de déclenchement d'une acquisition ;
2. un état **wait_first_edge** : afin de garantir la cohérence des données, le process attends un premier front du signal étudié avant de démarrer réellement l'acquisition ;
3. le dernier état **acquire_time** attend également le niveau haut signifiant la détection d'un front. Quand cette condition est évaluée comme vraie, une écriture dans la RAM est déclenchée. Dans le même temps, l'adresse est évaluée pour déterminer si l'acquisition est finie. Dans le premier cas, l'état passe à nouveau à **IDLE**, dans le cas contraire, la valeur de l'adresse est incrémentée.

8.5 Couche de communication côté FPGA

Le code du bloc de gestion de la communication, proposé dans ce projet ISE, est celui de l'exercice précédent et doit donc être modifié afin de changer la taille du bus de donnée de la RAM (32 bits maintenant). Comme le bus **wbs_readdata** ne peut pas être modifié (taille imposée par le matériel), il est donc nécessaire d'ajouter un registre et de découper le transfert d'une valeur en deux transactions. **Attention** : l'incrément de l'adresse ne doit être faite qu'une seule fois (soit lors de l'accès à la partie haute du mot ou à sa partie basse) et attention également à accéder à ce registre en second.

8.6 Communication côté CPU

Là encore, compte tenu de la ressemblance de cet exercice avec le précédent, le code est celui de l'exercice précédent, il faut donc l'adapter pour tenir compte de la taille des données et du mode de récupération de celles-ci. Deux lectures de 16 bits (`unsigned short`) doivent être faites, puis le mot de 32 bits reconstruits par concaténation et décalage binaire (ie `result = valH << 16 | (valL & 0xffff);`.

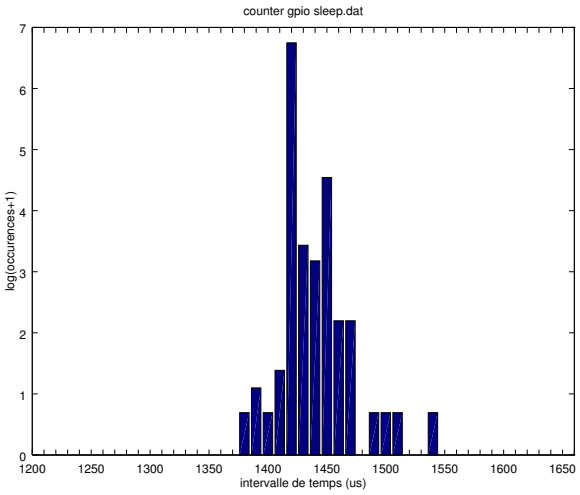
8.7 Mesure de la période du signal étudié

Les résultats pour *gpio_sleep* (Figs. 10, *xeno_gpio_sleep* (Figs. 11) et *xeno_gpio_timer* (Figs. **??**) présentent des résultats cohérents avec les mesures déjà décrites à l'aide d'un oscilloscope. Les mesures pour l'application *gpio_sigalarm* ne sont pas présentées car semblent très suspectes (dispersion des mesures excessive).

Les histogrammes ont étés obtenus avec octave grâce aux commandes :

```
a = load('nom_du_fichier.dat');
a = a./100; % conversion base 10ns en 1us
figure;[xx,nn]=hist(a,[1150:100:2000]);bar(nn,log(xx+1))
xlim([1150 2000]);xlabel('intervalle de temps (us)');ylabel('log(occurences+1)')
title('un titre');
```

Les limites `[1150:100:2000]` sont à adapter en fonction des mesures.



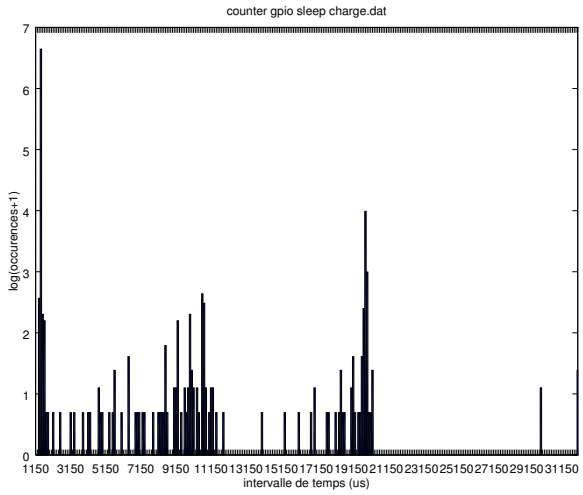
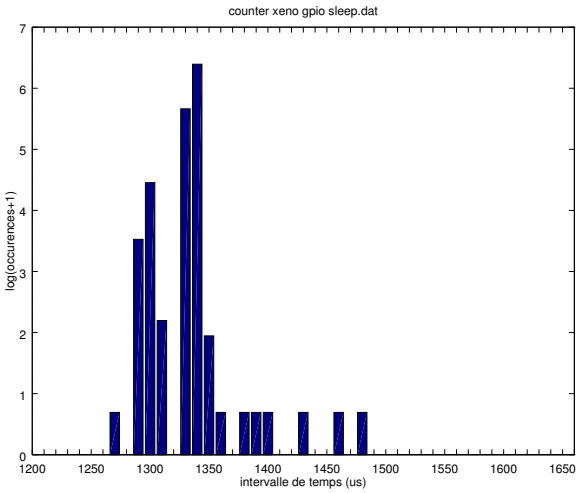


FIGURE 10 – Mesure d'un signal périodique généré par une application en espace utilisateur dans laquelle le changement d'état de la sortie intervient au bout de $500\mu s$ par appel à la commande **usleep**, donnant ainsi une période de $1ms$. À gauche sur une système non chargé la période dure en moyenne 1440 μs (min 1377 μs, max 1538 μs). À droite, sur une système chargé, le signal fluctue entre 1385 μs et 40 ms.



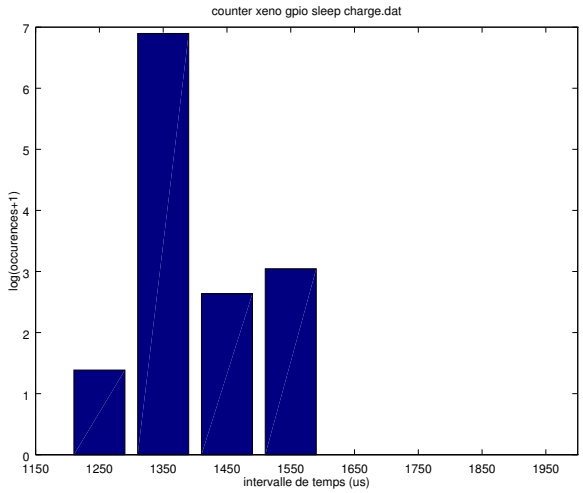


FIGURE 11 – Xenomai : génération d'un signal de 1 ms sur attente. À gauche, système non chargé, les périodes ont majoritairement une durée de 1340 μs (min : 1271 μs, max : 1476 μs). À droite, la tendance reste équivalente (min : 1251 μs, max : 1560 μs).

9 Générateur de signaux périodiques

Nous avons vu que GNU/Linux, du fait de son aspect multitâches préemptif et temps partagé, n'est pas adapté à des traitements à fortes contraintes temporelles. La solution temps-réel dur, proposée par Xenomai, offre des performances plus acceptables mais reste toutefois du temps-réel logiciel. En effet une telle solution permet de garantir des retards de quelques dizaines de μs.

Dès lors que la contrainte temporelle devient primordiale et que les traitements ne peuvent souffrir de jitters, une des seules solutions envisageables est l'utilisation d'un FPGA.

Pour illustrer ce cas de figure, le sujet de cet exercice est l'implémentation d'un générateur de PWM (*Pulse Wave*

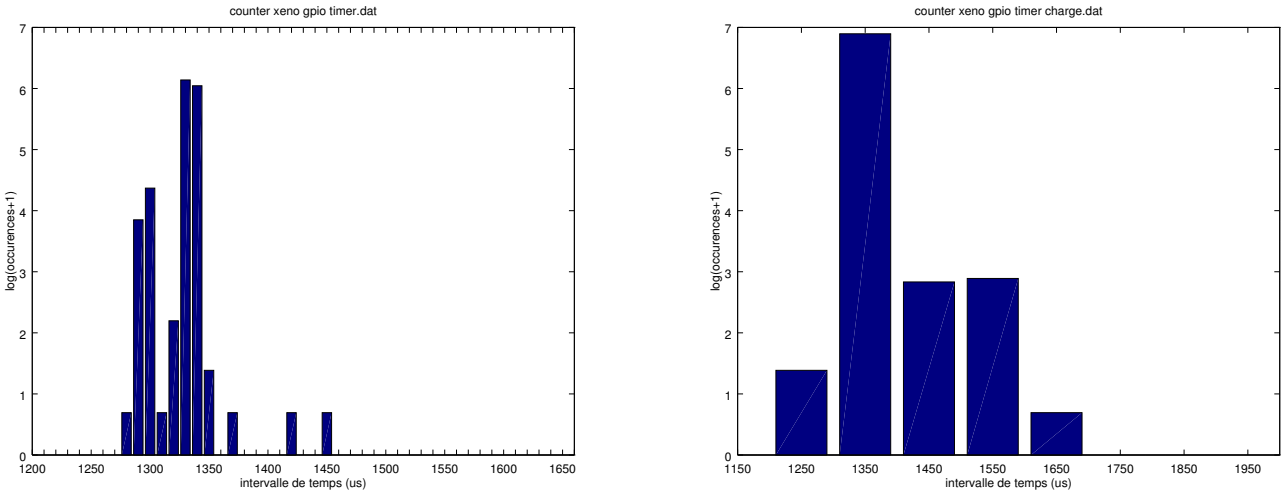


FIGURE 12 – Xenomai : génération du signal période de $1ms$ sur timer. À gauche, système non chargé les périodes ont majoritairement une durée de 1330 μs (min : 1281 μs, max : 1451 μs). À droite, la tendance reste équivalente (de 1256 μs à 1623 μs).

Modulation). Ce type de module, que l'on trouve dans certains microprocesseur et micro-contrôleurs permet de générer un signal périodique dont il est possible d'ajuster la période mais également le rapport cyclique (rapport entre la durée de l'état haut et de l'état bas) ou durée de la demi-période.

Le sujet de cet exercice est, en se basant sur les codes déjà réalisés, de créer un bloc, dont les paramètres de périodes et de rapport cyclique sont contrôlables depuis le processeur, avec pour base de temps la μs.

Pour cela il faut proposer une solution pour incrémenter le compteur à une fréquence plus faible que celle du FPGA et gérer les deux cas demi périodes.

Références

[1] *Wishbone B4 – WISHBONE System-on-Chip (SoC)Interconnection Architecture for Portable IP Cores*, OpenCores, 2010