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

G. Goavec-Merou, 17 janvier 2019

Sujet disponible sur `http://www.trabucayre.com/enseignement/tp_fpga.pdf`
Codes disponible sur `http://www.trabucayre.com/enseignement/tp_fpga_sources.tgz`

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 `Redpitaya`, à base de `Zynq7000`, 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 latences – 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 8 ns, puisque le cœur du FPGA est cadencé à 125 MHz [1]

Le présent TP s'articule de la façon suivante :

1. la première étape va consister à s'approprier le principe de communication CPU/FPGA et le bus `AXI` ;
2. ensuite une reprise du TP sur Xenomai sera nécessaire pour disposer des applications en espace utilisateur nécessaires pour la dernière partie du TP ;
3. finalement nous allons réaliser un compteur de période, dans le FPGA, afin d'évaluer les caractéristiques de stabilité à la charge d'applications Linux et Xenomai.

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

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 :

```
1 make
```

Pour les applications temps-réelles la commande sera :

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

2.2 Obtenir un terminal sur la carte

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

```
1 screen /dev/ttyUSB0 115200
```

2.3 Accès aux binaires depuis la plate-forme

Au lieu de transférer les fichiers sur la carte `Redpitaya` 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).

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

```
1 # mount 192.168.0.1:/home/etudiant/nfs /mnt
```

Avec :
— *192.168.0.1*, l'adresse correspondant à l'IP du PC (à adapter à chaque poste) ;
— */home/etudiant/nfs* le répertoire du PC qui doit être monté ;
— */mnt* l'endroit, dans l'arborescence du système de fichier de la carte, où il sera monté.

À partir de maintenant tous les binaires (.bit.bin pour le FPGA et applications) devront être copiés dans le répertoire */home/etudiant/nfs* du PC et l'ensemble du travail sur la carte devra être fait dans le répertoire */mnt*

1. on prendra cependant soin de noter que ce 125 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.

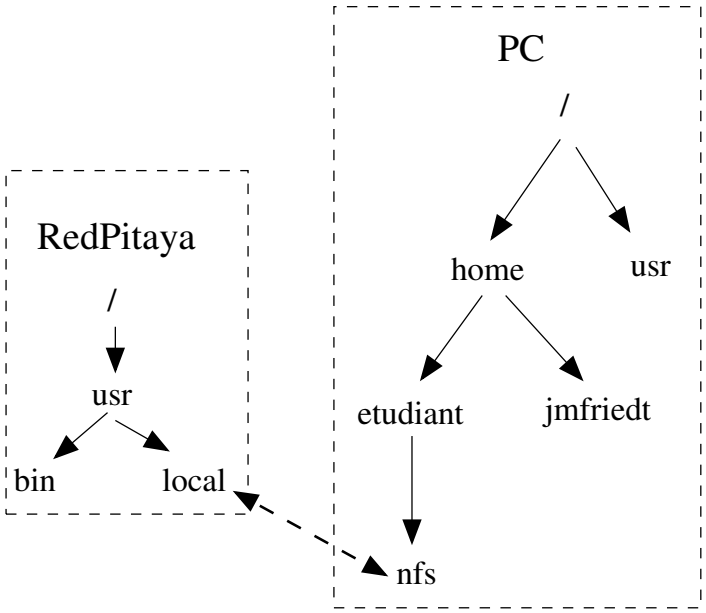


FIGURE 1 – Principe de l'utilisation de NFS : Le contenu du répertoire /home/etudiant/nfs du PC est accessible directement depuis la carte cible au travers du répertoire /mnt.

3 Communication CPU-FPGA

Nous avons pu voir, dans la première partie du TP, les caractéristiques temporelles de Linux et de Xenomai, les mesures ont pu 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.

4 Environnements matériel et logiciel

Comme présenté plus tôt, le matériel utilisé, une carte `RedPitaya`, basé sur un `Zynq` comporte dans une même puce un processeur généraliste et un FPGA. La communication entre les deux zones se fait par des bus `AXI` (fig 2).

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.

Les `design` qui seront utilisés dans la suite, se différencient de l'approche classique du développement sur `Zynq`. Ceci pour éviter la complexité des mises à jour des `package` à chaque modification du code. Les diverses applications ont étés générés à l'aide de *Peripheral On Demand*[2] qui est un outil d'assemblage de blocs HDL (non traité dans ce TP). Le `bloc design` contenant le `processing system` est confiné et le bus `AXI` exporté. `POD` génére un décodeur d'adresse (l'`intercon` pour communiquer de manière indépendante avec chaque esclave (figure 3) intégré dans le `design`(figure 4). Le rôle de ce composant est de découper la plage mémoire en plusieurs zones.

4.1 Génération du binaire

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

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

```
source /opt/Xilinx/Vivado/VERSION/settings64.sh
```

(Attention, les applications lancées depuis ce terminal peuvent présenter des erreurs d'exécutions).

De taper la commande

```
vivado&
```

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

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

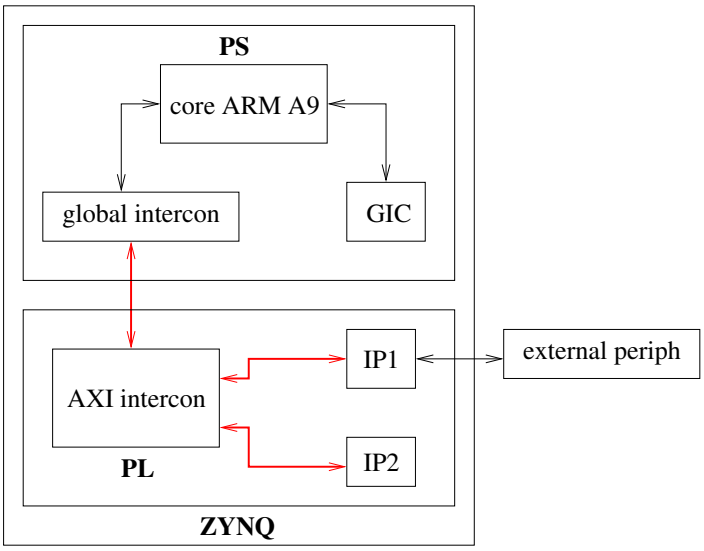


FIGURE 2 – Schéma simplifié de la structure interne d'un `ZYNQ`.

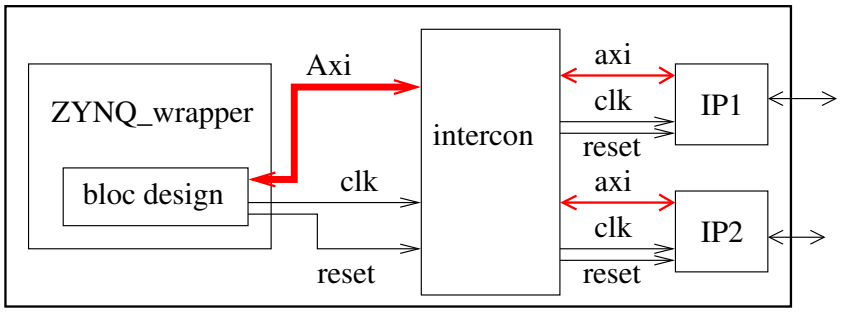


FIGURE 3 – Exemple de design FPGA

5 Découverte de la communication FPGA-CPU : addition

Le projet VHDL se trouve dans le répertoire /home/etudiant/tp_fpga_sources/design/exercice_addition et le fichier à ouvrir dans `vivado` dans le sous-répertoire objs/exercice_addition.xpr

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

5.1 Principe de base de la communication `AXI`

Compte tenu de la relative complexité du bus `AXI` (fig. 5) et la couche d'abstraction présenté précédemment, la partie dédiée à la gestion de la communication ne doit prendre en compte qu'un sous-ensemble de signaux (fig. 6). Hormis les deux bus de données `s00_axi_wdata` et `s00_axi_rdata`, les autres signaux sont créés localement et utilisés entre la couche d'abstraction et les `process`.

La gestion des transactions avec le processeur présentent deux signaux de contrôles :

- `write_en_s` : ordre d'écriture (du point de vus processeur), actif à l'état haut ;
- `read_en_s` : ordre de lecture, actif, également, à l'état haut ;

Deux signaux de données et un commun pour les adresses :

- `addr_s` : adresse du registre, commun à l'écriture et à la lecture ;
- `readdata_s` : bus de donnée pour les requêtes de lectures. Doit être mis à jour par l'IP. Ce signal local est nécessaire car il doit être maintenu à jour avec sa propre valeur ce qui n'est pas possible avec le le signal `s00_axi_rdata` qui est exclusivement en écriture (mode `out` au niveau de l'entité) ;
- `s00_axi_wdata` : bus de donnée pour les requêtes d'écriture. Est lu par l'IP.

5.2 Implémentation du bloc d'addition

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

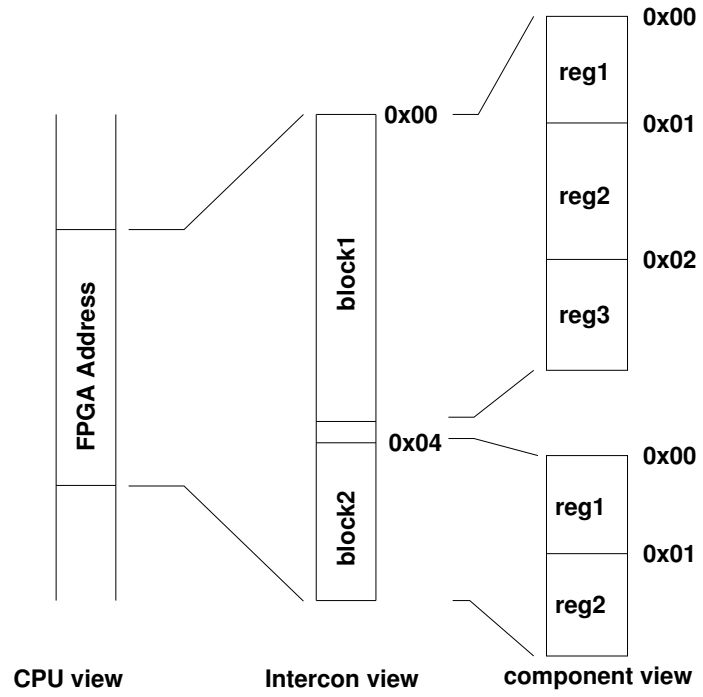


FIGURE 4 – Principe d'abstraction des adresses des composants dans le FPGA. La plage d'adresse dédiée à la communication entre le CPU et le FPGA est découpée en plusieurs sous-adresses. L'intercon réalise le décodage et adresse le composant visé en lui fournissant le numéro du registre correspondant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

```
Entity enseignement_addition_axi is
    generic(
        id          : natural := 1;
    -- Parameters of Axi Slave Bus Interface S00_AXI
    C_S00_AXI_DATA_WIDTH   : integer := 32;
    C_S00_AXI_ADDR_WIDTH   : integer := 5);
    port (
    -- axi slave signals
    s00_axi_aclk   : in std_logic;
    s00_axi_reset  : in std_logic;
    s00_axi_awaddr   : in std_logic_vector(C_S00_AXI_ADDR_WIDTH-1 downto 0);
    s00_axi_awprot   : in std_logic_vector(2 downto 0);
    s00_axi_awvalid  : in std_logic;
    s00_axi_awready  : out std_logic;
    s00_axi_wdata  : in std_logic_vector(C_S00_AXI_DATA_WIDTH-1 downto 0);
    s00_axi_wvalid   : in std_logic;
    s00_axi_wready   : out std_logic;
    s00_axi_wstrb  : in std_logic_vector(3 downto 0);
    s00_axi_bresp  : out std_logic_vector(1 downto 0);
    s00_axi_bvalid   : out std_logic;
    s00_axi_bready   : in std_logic;
    s00_axi_araddr   : in std_logic_vector(C_S00_AXI_ADDR_WIDTH-1 downto 0);
    s00_axi_arprot   : in std_logic_vector(2 downto 0);
    s00_axi_arvalid  : in std_logic;
    s00_axi_arready  : out std_logic;
    s00_axi_rdata  : out std_logic_vector(C_S00_AXI_DATA_WIDTH-1 downto 0);
    s00_axi_rresp  : out std_logic_vector(1 downto 0);
    s00_axi_rvalid   : out std_logic;
    s00_axi_rready   : in std_logic);
end entity enseignement_addition_axi;
```

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

```
Architecture enseignement_addition_1 of enseignement_addition_axi is
  signal addr_s : std_logic_vector(INTERNAL_ADDR_WIDTH-1 downto 0);
  signal write_en_s, read_en_s : std_logic;

  signal readdata_s : std_logic_vector(C_S00_AXI_DATA_WIDTH-1 downto 0);
begin
  s00_axi_rdata <= readdata_s;
```

FIGURE 6 – Sous-ensemble de signaux nécessaires pour la gestion des accès depuis le processeur.

```
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`.

5.3 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
    readdata_s <= (others => '0');
  elsif rising_edge(clk) then
    readdata_s <= readdata_s;
    if read_en_s = '1' then
      case addr_s is
      when REG_ID =>
        readdata_s <= std_logic_vector(to_unsigned(id, dat_size));
      when REG_RESULT =>
        readdata_s <= result_s;
      when others =>
        readdata_s <= (others => '0');
      end case;
    end if;
  end if;
end process read_bloc;
```

Dans ce `process` nous voyons que nous vérifions l'état du signal `read_en_s` qui passe, pendant un cycle d'horloge, à l'état haut lors d'une requête de lecture. Si la condition n'est pas remplie, la valeur actuelle de `readata_s` est maintenu. Dans le cas contraire, lors d'un ordre de lecture, l'étape suivante est d'évaluer `addr_s`, avec un `case` pour connaître quel registre est accédé. Si l'adresse ne correspond pas à un registre valide (donc ni à `REG_ID`, ni à `REG_RESULT`), nous affectons une valeur quelconque (ou rien) à `readdata_s`. Dans le cas d'un registre valide, la valeur correspondante est affectée au registre de lecture.

5.4 Gestion de l'écriture

Pour une requête d'écriture le principe est sensiblement identique à la lecture : nous évaluons le signal `write_en_s` qui est le pendant, pour l'écriture, de `read_en_s`.

```
write_bloc : process(clk, reset)
begin
   if reset = '1' then
     op1_s <= (others => '0');
     op2_s <= (others => '0');
   elsif rising_edge(clk) then
     op1_s <= op1_s;
     op2_s <= op2_s;
    if write_en_s = '1' then
```

```
      case addr_s is
      when REG_OP1 =>
        op1_s <= writedata;
      when REG_OP2 =>
        op2_s <= writedata;
      when others =>
      end case;
      end if;
   end if;
end process write_bloc;
```

Là encore nous testons, au travers d'un `case` à quel registre correspond la requête. Si l'adresse correspond à un registre valide, la valeur de `writedata` est affecté au signal correspondant. Dans le cas d'une requête invalide, rien n'est réalisé.

5.5 Communication avec le FPGA

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

```
devmem 0x10 32 --> read @ 0x10
devmem 0x10 32 0x1234 --> writes 0x1234 @ 0x10
```

le seconde paramètre *32* signifie des accès en 32 bits.

L'adresse de base de notre IP est 0x43C00000.

Deux points importants à prendre en compte :

1. l'adresse utilisée par **devmem** est absolue ;
2. la communication se fait en 32 bits (soit 4 octets), donc passer d'un registre au suivant se fait en incrémentant de 4. Ou d'une autre mesure accéder à un registre nécessite de réaliser un décallage de 2 vers la gauche (multiplication par 4) pour aligner les accès sur des adresses 32 bits ;
3. le second paramètre (la taille) est optionnelle en cas de lecture. Dans le cas où il n'est pas fournie l'accès est sur 32 bits. Il est toutefois obligatoire pour l'écriture.

Lecture de l'ID du bloc (adresse 0x43C00000 + 0x00) :

```
redpitaya> devmem 0x43C00000
0x00000001
```

Écriture de la valeur 2 dans le registre **REG_OP1** (0x43C00004 = 0x43C00000 + 0x01 << 2)

```
redpitaya> devmem 0x43C00004 32 2
```

Écriture de la valeur 3 dans le registre **REG_OP2** (0x43C00008 = 0x43C00000 + 0x02 << 2)

```
redpitaya> devmem 0x43C00008 32 3
```

Lecture du résultat (5) depuis le registre **REG_RESULT** (0x43C0000c = 0x43C00000 + 0x03 << 2)

```
redpitaya> devmem 0x43C0000c
0x00000005
```

5.6 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, la valeur ;
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;
```

6 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 vivado est dans le sous-répertoire objs/exercice_ram.xpr

L'idée de cette seconde application est de lancer un traitement long à partir du processeur. Ce traitement sera simulé par le remplissage d'une RAM avec des données arbitraire. Le bloc devra, par ailleur, fournir un statut pour que le processeur puisse déterminer s'il doit encore attendre avant de réaliser la récupération des résultats.

Notre second exercice sera de réaliser un bloc qui :

1. suite à la réception d'un ordre de `start` du processeur, lance 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 statut 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 :

31	...	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ID															

31	...	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X															busy

31	...	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X															start

31	...	13	12	11	10	9	8	7	6	5	4	3	2	1	0
data															

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

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

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

registre 0x03 : lecture seule. Utilisé pour récupérer les valeurs successives 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.

6.1 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 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;
        cpt_addr_s <= (others => '0');
      end if;
    else -- state storage
      cpt_addr_s <= std_logic_vector(unsigned(cpt_addr_s) + 1);
      cpt_data_s <= (31 downto 10 => '0') & cpt_addr_s;
      cpt_en_s <= '1';
      if cpt_addr_next_s = 1023 then
        busy_s <= '0';
```

```
      else
26      cpt_addr_next_s <= cpt_addr_next_s+1;
      end if;
28    end if;
  end if;
30 end process;
```

6.2 Communication AXI : synthesis/enseignement_ram/ens_ram_comm.vhd

6.2.1 Gestion de l'écriture

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.

Le signal géré par **REG_START** ne doit pas conserver sa valeur : au prochain cycle d'horloge, dès lors que les conditions d'écriture ne sont plus validées, le signal est remis au niveau bas. Ceci est nécessaire pour éviter que l'acquisition ne se lance en boucle.

6.2.2 Gestion de la lecture

En plus de gérer l'id il doit gérer :

— les requêtes pour connaître l'état du bloc (occupé ou inactif). Ce traitement consiste à concaténer un ensemble de bit à l'état bas ((31 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 successives contenues dans la RAM. 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.

Remarque : pour la lecture des données, le `process`, doit à chaque requêtes, réaliser deux opérations :

1. incrémenter l'adresse de la RAM ;
2. affecter au signal `readdata_s` la valeur courante du bus de données de la RAM.

6.3 Communication depuis le processeur

Des outils comme *devmem* ne sont pas adaptés pour gérer un nombre important d'accès. Il serait bien entendu possible de réaliser un script shell pour simplifier mais il faut prendre en compte que pour chaque appel l'outil passe par un ensemble d'étapes pour obtenir un accès à la mémoire.

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);
2 if (fd < 0) {
  printf("can't open file /dev/mem\n");
4 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 :

```
1 void *ptr_fpga = mmap(0, 8192, PROT_READ|PROT_WRITE, MAP_SHARED,
        fd, 0x43C00000);
3 if (ptr_fpga == MAP_FAILED) {
  printf("mmap failed\n");
5 return -2;
}
```

et finalement réaliser des lectures de la manière suivante :

```
1 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;
2 *(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, page_size, PROT_READ|PROT_WRITE, MAP_SHARED,
            fd, FPGA_BASE_ADDR);
if (ptr_fpga == MAP_FAILED)
  return -2;

pos = FPGA_OFFSET + REG_ID;
value = *(unsigned short*)(ptr_fpga+((unsigned short)pos));
if (value != 1)
  return EXIT_FAILURE;

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

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

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.14-15) et attends la fin de celle-ci(l.17-20) ;
— pour finir récupére séquentiellement l'ensemble des valeurs contenues dans la RAM (l.22-26) et les affiche 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* à l'aide de la commande **cp**.

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

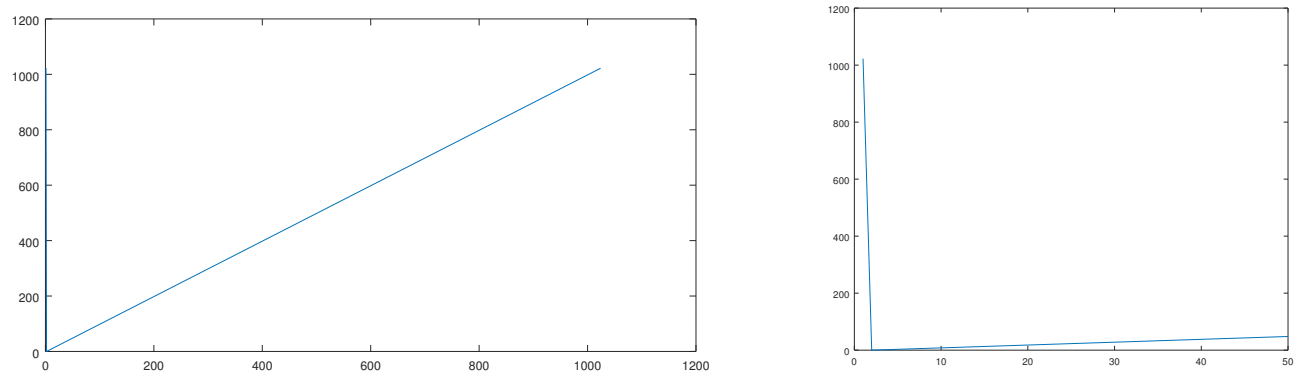


FIGURE 7 – Le résultat semble bon (figure de gauche) ... Mais présente une incohérence (la rampe semble décalée d'un échantillon).

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 7.

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

Nous pouvons constater qu'au lieu d'avoir une suite entre 0 et 1023 (soit 1024 points), le premier point vaut 1023 puis la suite est logique avec 1022 comme dernière valeur.

Proposez une solution pour que les données soient correctement alignées (l'erreur ne viens pas forcément de la partie communication).

7 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 d'une communication par bus `AXI`. 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.

7.1 Implémentation du compteur de durée de périodes

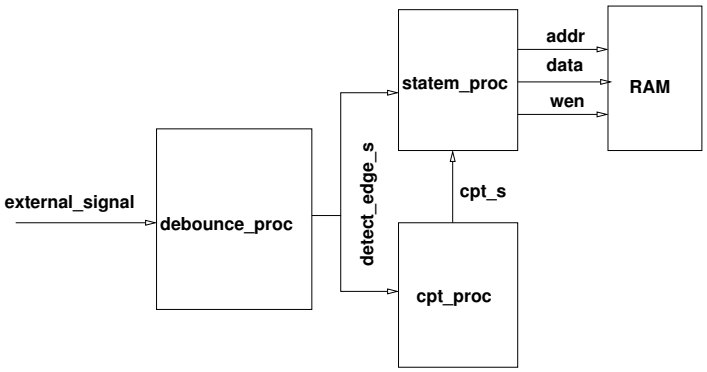


FIGURE 8 – Schéma global du traitement à implémenter

La partie compteur de ce composant est découpé en trois `process` (Fig. 8) :

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 cet ordre est reçu, le `process` réalise n acquisition consécutives avant de se remettre dans l'état initial.

7.2 debounce_proc

Le principe de ce `process` est d'évaluer une série d'états passés du signal à étudier pour s'assurer que celui-ci 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`.

7.3 cpt_proc

Ce `process` sonde, à chaque cycle d'horloge l'état du signal de détection d'un 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 bas 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.

7.4 statem_proc

Globalement ce `process` est proche de la pseudo-acquisition de l'exercice précédent.
Il comporte trois états :

1. un état **IDLE** : le `process` est en attente de l'ordre, venant du processeur, de déclenchement d'une série acquisitions ;

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.

7.5 Couche de communication côté FPGA

Le code du bloc de gestion de la communication, proposé dans ce projet, est celui de l'exercice précédent.

7.6 Communication côté CPU

Là encore, compte tenu de la ressemblance de cet exercice avec le précédent, le code est celui de l'exercice précédent.

7.7 Mesure de la période du signal étudié

Les résultats pour *gpio_sleep* (Figs. 9, *xeno_gpio_sleep* (Figs. 10) 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./125; % conversion base 8ns 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.

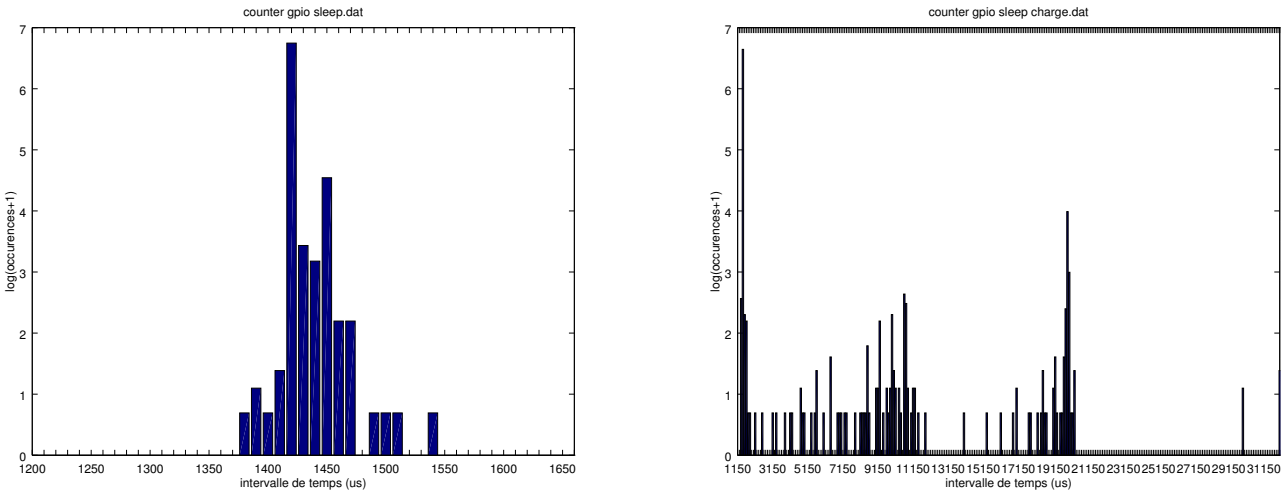


FIGURE 9 – Mesure d'un signal périodique généré par une application en espace utilisateur dans laquelle le changement d'état de la sortie intervient au bout de $500\mu s$ par appel à la commande **usleep**, donnant ainsi une période de $1ms$. À gauche sur une système non chargé la période dure en moyenne 1440 μs (min 1377 μs, max 1538 μs). À droite, sur une système chargé, le signal fluctue entre 1385 μs et 40 ms.

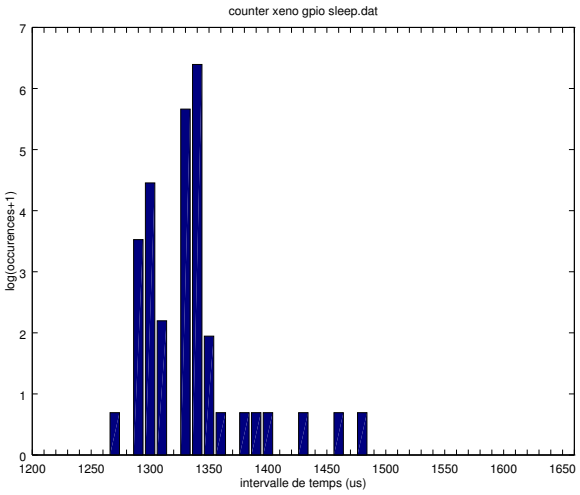
8 Qualification Linux et Xenomai

L'ensemble des applications se trouve dans le répertoire */home/etudiant/tp_fpga_sources/apps/gpio_test_xeno*

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 `E8/PS_MIO13_500` correspondant à la pin 6 du connecteur E2 de la carte `Redpitaya`) 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.



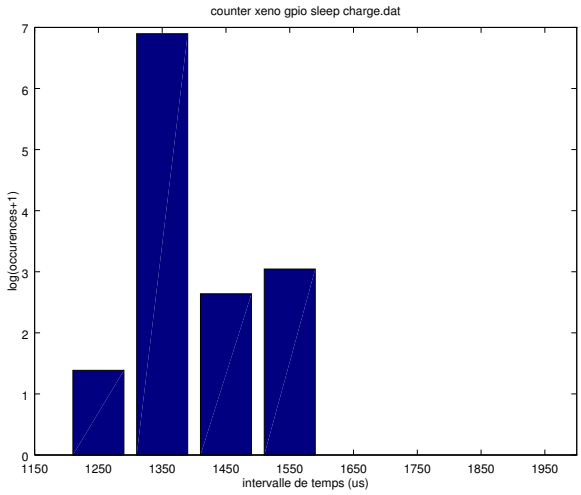
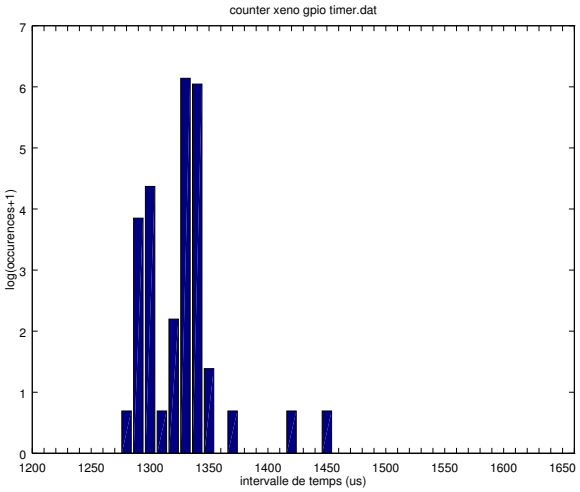


FIGURE 10 – Xenomai : génération d'un signal de 1 ms sur attente. À gauche, système non chargé, les périodes ont majoritairement une durée de 1340 μs (min : 1271 μs, max : 1476 μs). À droite, la tendance reste équivalente (min : 1251 μs, max : 1560 μs).



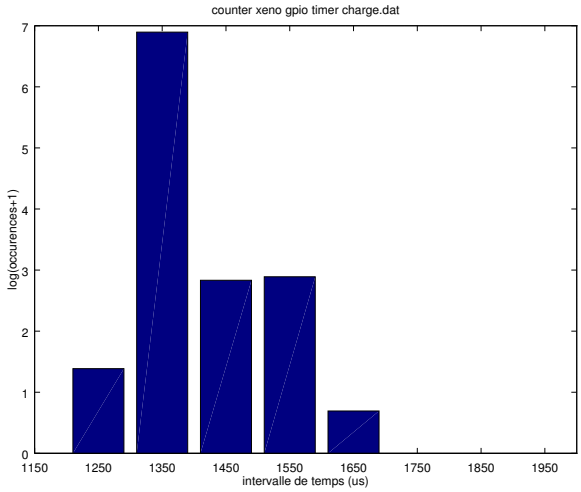


FIGURE 11 – Xenomai : génération du signal période de $1ms$ sur timer. À gauche, système non chargé les périodes ont majoritairement une durée de 1330 μs (min : 1281 μs, max : 1451 μs). À droite, la tendance reste équivalente (de 1256 μs à 1623 μs).

8.1 Génération d'un signal sur une mise en sommeil

```
int main(void)
{
    configuration_gpio;

    while(1) {
        changement_etat_broche
        mise_en_sommeil
    }
}
```

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

```

void blink(void *arg)
{
    while(1) {
        changement_etat_broche
        mise_en_sommeil
    }

}

int main(void)
{
    configuration_gpio;
    creation_task
    demarrage_task
    attente
    destruction_task

}
```

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

8.1.1 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 :

```
unsigned char *red_gpio_init(int pin_num);
```

avec `pin_num` le numéro de la broche utilisée. Elle retourne un pointeur sur la zone mémoire du contrôleur de gpio.
Puis configuration de la broche en sortie :

```
void red_gpio_set_cfgpin(unsigned char *mem, int num);
```

avec :
- **unsigned char *mem** correspond au pointeur retourné par la fonction `reg_gpio_init` ;
- **num** le numéro de la broche, dans le cas présent 13.

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

```
void red_gpio_output(unsigned char *mem, int num, int value);
```

avec :
- là encore **mem** correspond au pointeur sur la mémoire ;
- **num** à la broche ;
- **value** à l'état que doit prendre la broche. Une valeur non nulle pour mettre la broche à l'état haut, 0 pour la passer à l'état bas.

8.1.2 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 *u*s
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**.

8.1.3 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()*

8.2 Génération d'un signal sur timer

```
1
2  void blink(int signum)
3  {
4      changement_etat_broche
5  }
6
7  int main(void)
8  {
9      configuration_gpio;
10
11     configuration_gestionnaire_event
       configuration_timer
12
13     while(1) {}
14 }
15
```

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

```
1
2  void blink(void *arg)
3  {
4      configuration_timer
5
6      while(1) {
7          attente_timer
8          changement_etat_broche
9      }
10 }
11
12 int main(void)
13 {
14     configuration_gpio;
15     creation_task
16     demarrage_task
17     attente
18     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.

8.2.1 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 initialialisée 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.

8.2.2 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.

8.3 Utilisation

8.3.1 Compteur de période

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

Les applications préalablement copiées dans `/home/etudiant/nfs` sont disponibles depuis la plate-forme dans le répertoire `/mnt`

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

```
cp top_exercice_counter.bit.bin /lib/firmware
echo top_exercice_counter.bit.bin > /sys/class/fpga_manager/fpga0/firmware
```

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

```
./app_exercice_counter
```

8.3.2 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 créé 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_sources/apps/app_exercice_counter*. Pour le lancer :

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

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

8.3.3 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 8ns => /125 pour us
a=a./125;
% 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.

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 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