Instrumentation scientifique reconfigurable

T. Rétornaz, J.-M. Friedt

12 mai 2008

Nous proposons d'explorer pas à pas l'utilisation d'une carte comprenant un FPGA et un processeur pour le développement de systèmes de mesures reconfigurables. Nous allons appliquer les compétences acquises à la réalisation et caractérisation d'un compteur de fréquence pour la mesure d'oscillateurs. Une illustration plus complexe des performances du couple FPGA+processeur sera l'acquisition d'images et affichage sur écran LCD en tentant de décharger au maximum le processeur par l'exploitation du FPGA comme module autonome d'acquisition.

1 Introduction

La capacité à reconfigurer une matrice de portes logiques (FPGA [1]) selon les besoins de l'utilisateur ouvre de vastes possibilités dans de nombreux domaines, qu'il s'agisse d'informatique avec des processeurs reconfigurables selon les tâches exécutées [1], ou de matériel qui s'adapte aux demandes de l'utilisateur [2] (configuration du FPGA comme coprocesseur vidéo [3], traitement du signal ou pour des opérations de calcul flottant par exemple [4, 5]). Nous allons ici nous intéresser à l'utilisation de FPGA à des fins de prototypage d'instruments de mesure scientifique. L'intérêt du FPGA consiste alors dans le coût réduit du matériel nécessaire lors de la phase de développement (notamment en comparaison avec la réalisation sur silicium d'un composant avec les mêmes fonctionnalités) et à la compatibilité des outils de développement avec la fabrication de circuits dédiés sur silicium (ASIC [2]).

Nous allons pour ces tests utiliser la plateforme de développement proposée par l'association Armadeus Project [6] (nommée Armadeus ultérieurement), qui associe la souplesse du FPGA à un processeur ARM9 [7] fonctionnant sous GNU/Linux, et grâce auquel nous pourrons donc rapidement contrôler notre instrument, récupérer et traiter des informations, pour ensuite les communiquer à l'utilisateur.

2 Outils de développement sur Armadeus

La carte DevLight proposée par Armadeus inclut les divers connecteurs (difficiles à trouver par ailleurs) pour exploiter pleinement la carte mère dénommée APF9328 que nous utiliserons au cours des nos développements. Cette dernière inclut un processeur, basé sur une architecture ARM9, cadencé à 200 MHz (comportant lui même un grand nombre de périphériques embarqués, les seuls dont nous ferons usage étant le contrôleur ethernet, l'UART pour la communication RS232, et l'interface de communication I^2C), et un FPGA Xilinx Spartan3 de 200 kportes.

L'utilisation de l'architecture Xilinx a son importance quant à la disponibilité d'outils gratuits (à défaut d'être libres) de développements, de consommation et de vitesse de fonctionnement. En effet, un des inconvénients majeurs des FPGA pour les applications embarquées est leur consommation élevée.

[1] *Field-Programmable Gate Array* est un composant électronique, reconfigurable par logiciel, contenant typiquement quelques centaines de milliers de fonctions logiques. Ces fonctions sont agencées selon une description fonctionnelle du composant dans des langages évolués tels que VHDL ou Verilog.

[2] *Application Specific Integrated Circuit* est un composant électronique développé spécifiquement pour exécuter une tâche très spécifique, généralement en vue d'une réduction de coût par rapport à une solution utilisant un composant générique (microcontrôleur par exemple) programmé.

2.1 Installation des outils de développement pour ARM9

Une *toolchain* est l'ensemble des outils nécessaires à la compilation de binaires à destination d'une architecture :

- la chaîne compilateur-éditeur de liens tenant compte de l'agencement de la mémoire sur la plateforme cible : le trio *gcc*, `binutils` et `newlib` sont généralement nécessaires pour compiler une toolchain fonctionnelle,
- éventuellement un debugger si la plateforme cible le supporte : sous GNU/Linux, il s'agira généralement du couple client et serveur de `gdb`
- les librairies associées afin de fournir les outils de base (`libc` dans le cas de `gcc`) ainsi que diverses librairies annexes répondant aux besoins du développeur.

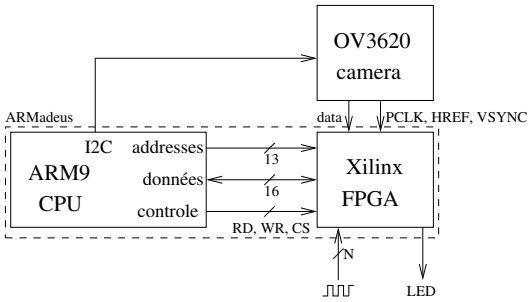


FIG. 1 – La carte APF9328 de Armadeus sera le cœur des expériences présentées dans ce document. Elle intègre un processeur basé sur un processeur ARM9 et un FPGA puissant reliés par des bus de communication. Nous ajoutons des périphériques afin d'illustrer diverses utilisation du FPGA comme instrument reconfigurable : une diode électroluminescente (LED), un capteur d'image CMOS pour l'imagerie et des signaux radiofréquences pour le développement de compteurs rapides.

Nous désirons développer un certain nombre de programmes pour communiquer entre un PC et la carte Armadeus, ou entre l'ARM et le FPGA. Il nous faut donc les outils pour compiler des applications à destination d'un ARM9 sur un PC (probablement basé sur une architecture Intel). Parmi les excellentes pages mises à disposition par l'association Armadeus [6], celle de l'installation de la toolchain est particulièrement claire : `http://www.armadeus.com/wiki/index.php?title=LinuxInstall`. Noter que le développement d'une toolchain à destination d'un nouveau circuit embarqué est une tâche fastidieuse car au-delà de la simple installation du cross-compilateur, il faut renseigner ce dernier sur les plages d'adresses accessibles par le processeur et dans lesquelles le compilateur a le droit de placer le code, la pile, le tas etc ... Une connaissance précise du matériel est donc nécessaire pour accomplir ces opérations. Une description de ce type d'activités est par exemple décrit à `http://uldp.home.at/uclinux_doc_4.html` : le script de linkage (généralement avec une extension `.ld` ou `.x`) est donné comme argument à `gcc` par l'option `-T fichier.ld`.

2.2 Outils de développement Xilinx pour GNU/Linux

Comme tout logiciel propriétaire distribué sous GNU/Linux, les outils de synthèse pour le FPGA sont distribués sous la forme d'une archive compressée qu'on se contentera de désarchiver au bon endroit, puis de rendre les binaires associés exécutables. Là encore, toute la procédure est décrite sur le wiki de Armadeus : `http://www.armadeus.com/wiki/index.php?title=ISE_WebPack_installation_on_Linux`. On peut s'attendre, comme ce fut le cas pour MapleV ou Mathematica il y a quelques années (passage de `libc5` à `glibc`), à des problèmes de compatibilités de librairies dynamiques dans le futur, mais pour le moment les binaires sont fonctionnels en l'état sur une distribution Debian/Etch.

Notre choix s'est porté sur la programmation du FPGA en VHDL pour diverses raisons : pérénité, réutilisation et maintenance d'une solution développée autour d'un langage de program-

mation (opposée à une solution de configuration par interface graphique), gestion du projet multi-utilisateurs avec utilisation d'outils tels que `diff` et `patch`, utilisation d'un langage bien documenté et dont la synthèse ne dépend *a priori* pas de l'architecture du composant cible, avec notamment la capacité d'utiliser le code ainsi validé sur un composant dédié sur silicium (ASIC). Une archive contenant tous les codes présentés dans ce document est disponible à http://jmfriedt.free.fr.

Bien qu'il soit techniquement possible d'installer une plateforme de développement et de simulation VHDL complètement libre sous licence GNU, il est toujours nécessaire de recourir à l'ISE (*Integrated Software Environment* propriétaire Xilinx lors de la synthèse du code en fin de conception du projet (http://www.armadeus.com/wiki/index.php?title=How_to_make_a_VHDL_design_in_Ubuntu/Debian). Nous utiliserons donc cet outils au cours de l'ensemble des développements présentés dans ce document (Fig. 2).

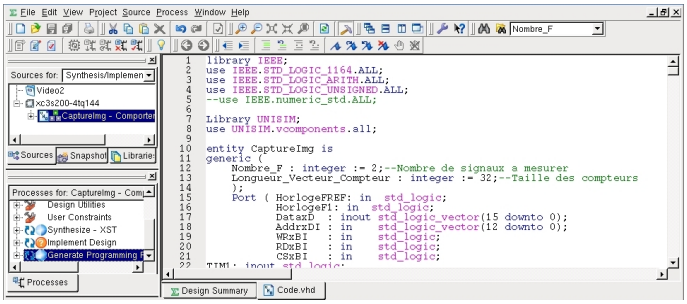


FIG. 2 – Capture d'écran du logiciel fourni par Xilinx, fonctionnel sous GNU/Linux, pour synthétiser le code VHDL à destination du FPGA Spartan3. Cette interface est exclusivement utilisée pour la phase de synthèse du code VHDL en fichier binaire servant à la configuration du FPGA.

3 Premiers pas sur FPGA

3.1 Contrôler une diode

Comme pour tout nouveau circuit embarqué que nous découvrons, notre premier objectif est de contrôler une diode électroluminescente (LED) connectée à une broche du FPGA. Il s'agit aussi d'une opportunité pour se familiariser avec les points d'accès aux broches du FPGA reportés sur la carte de développement DevLight, et d'apprendre à transférer un fichier de configuration d'un FPGA depuis la version de GNU/Linux tournant sur l'ARM9.

Nous allons donc commencer par nous familiariser avec un des atouts de la carte Armadeus, à savoir la liaison directe des bus de données et d'adresses entre l'ARM et le FPGA (Fig. 1). Cette communication parallèle à haut débit va être utilisée ici pour piloter l'alimentation d'une LED, connectée en sortie du FPGA. Comme sur tout bus, de nombreux signaux transitent continuellement à destinations des autres périphériques partageant ce support de communication : RAM, mémoire non-volatile (flash), contrôleur ethernet ... Afin de ne pas perturber la communication entre le processeur et ces périphériques, notre configuration sur le FPGA doit prendre soin de

- ne pas mobiliser le bus de données en configurant les broches associées en sortie (D0 à D15 du processeur sont reliées aux broches du banc 7 du FPGA tel que décrit dans http://www.armadeus.com/_downloads/apf9328/hardware/apf_schema.pdf), induisant un conflit avec les autres périphériques communiquant sur ce bus (broche en haute impédance ou entrée par défaut)
- ne pas mobiliser le bus d'adresses qui, en l'absence de DMA, impose sa valeur au FPGA qui n'a jamais à y écrire (broches en entrée)
- identifier lorsque le FPGA est appelé par un décodage d'adresse (lecture de la valeur sur le bus d'adresses et comparaison avec une plage assignée à notre composant programmable,

évitant tout conflit avec les périphériques installés par ailleurs sur la carte) associé à la gestion des signaux de contrôle issus du processeur (lecture ou écriture)
- conserver la valeur lue sur le bus de données une fois le cycle de communication achevé : une bascule (ou un latch) déclenchée par le décodeur d'adresse mémorise la donnée fournie sur le bus d'adresses pour traitement ultérieur.

On a donc affaire à 3 ensembles de signaux : le bus d'adresse qui identifie l'interlocuteur du processeur dans la discussion, le bus de donnée identifiant le contenu de la discussion, et le bus de contrôle identifiant la nature de la transaction (notamment écriture ou lecture pour les cas qui vont nous intéresser ici).

3.1.1 Code VHDL

Le plus simple pour démarrer avec la synthèse de code VHDL consiste à reprendre le code fourni en exemple dans le projet Armadeus (`BRAMTEST.vhdl`). On aura ainsi une trame dont on conservera la structure générale.

Le composant définissant la configuration du FPGA est inclus entre :

```
entity ComposantVHDL is ... end ComposantVHDL;
```

La description du composant inclut avant tout la définition des ports de communication en entrée ou en sortie (E/S) ainsi que leurs tailles : après le mot clé `Port(`, nous ajoutons la définition d'un signal sur un unique bit en sortie

```
LED: out std_logic;
```

à notre entité modélisée en VHDL. On y fera correspondre un état d'entrée (issu de la valeur du bus de données) et des états logiques en sortie : dans notre exemple il s'agira d'un état binaire : LED allumée, et LED éteinte. Un autre état possible du port est la haute impédance dans laquelle les broches n'imposent par leur état sur le bus, sans pour autant en traiter le contenu.

Vient ensuite la description de l'architecture du composant :

```
architecture Version01 of ComposantVHDL is ...
end Version01;
```

Cela regroupe les variables systèmes (`signal`), les entêtes d'E/S des composants VHDL génériques qui vont être intégrés au composant final (`component`), les constantes (numérique, et de connexion), et les fonctions événementielles (`process`).

La variable système qui nous intérresse est :

```
signal Etat_LED:std_logic;
```

: il s'agit d'une variable binaire, qui fera correspondre son état à celui de la sortie de la broche LED par

```
 LED <= Etat_LED;
```

Ceci représente une constante de connexion. Ce routage est en dehors de tout process, il sera donc opérationnel pendant toute la durée d'execution du code dans le FPGA.

Nous adaptons ensuite le process chargé du décodage d'adresse et de la gestion des données par le process :

```
process (RDxBI, WRxBI, ClkxCI, CSxBI, AddrxDI, DataxD, Registre, Compteur)
...
end process;
```

Ce process scrute les évènements sur les signaux de contrôle `CSxBI` (*chip select*) et `WRxBI` (sélection du mode écriture), ainsi que le bus d'adresses `AddrxDI`

```
if(WRxBI = '0' and CSxBI = '0') --ARM ecrit dans FPGA
... case AddrxDI is
```

afin de voir si l'ARM veut communiquer avec le FPGA au moyen du bus de données `DataxD`. En cas de validation du décodeur d'adresse

```
if WRxBI = '0' and CSxBI = '0' then--Linux ecrit
    case AddrxDI is
         when "0001011110100"    => --AddrxDI=0x02f4
              Etat_LED <= DataxD(0);
         when "0001011110000"    => --AddrxDI=0x02f0
...
```

l'état du bit de poids faible du bus de données (`Etat_LED<=DataxD(0);`) est recopié sur la broche contrôlant la LED.

Nous avons omis dans cet exemple de synchroniser les opérations sur un signal additionnel ClkxCI='1' afin de revenir au cas le plus général des processeurs qui valident la communication sur un décodage d'adresse, un bit de direction (ici `WRxBI`) et éventuellement un Chip Select lorsqu'il existe (`CSxBI`). Ce signal doit être correctement géré en cas de communication de valeurs 32 bits puisque la transaction de 2 mots de 16 bits se fait sur une seule transition de chip select mais deux transitions de l'horloge [8, FPGA timing diagrams].

Afin d'interfacer ce composant avec le reste du monde (ici le processeur ARM), il faut assigner chaque signal à des broches et configurer la direction de communication de chacune de ces broches. Ces opération se définissent dans le fichier d'extension `.ucf`.

Ce fichier fait correspondre le numéro d'une broche à sa désignation. Pour connaître le numéro de la broche, et sa position sur la carte DevLight, il faut recourir à la datasheet du Spartan 3 [9] et à la documentation de Armadeus (http://www.armadeus.com/_downloads/apf9328/hardware/apf_schema.pdf). Dans notre cas :

```
NET "LED"  LOC = "P112" | IOSTANDARD = LVCMOS33;
```

configure la broche (dénommée L01P_1) de contrôle de la LED.

Il faudra prendre soin, lors de la phase de communication du FPGA vers le processeur, de toujours avoir une condition par défaut qui place le bus de données en haute impédance et n'en impose le niveau que lorsque la logique d'écriture est validée

```
if RDxBI = '0' and CSxBI = '0' then -- Linux lit(FPGA)
 -- phase de lecture ...
else if WRxBI = '0' and CSxBI = '0' then -- Linux ecrit(FPGA)
 -- phase d'ecriture ...
else DataxD <= (others  => 'Z'); -- etat de haute impedance
```

En l'absence de cette précaution, nous créons un conflit sur le bus de données et de fournissons un opcode invalide au processeur.

Si nous suivons correctement les conseils sur la page FPGA du wiki Armadeus (bien activer la génération d'un `.bin` dans les propriétés du module tel que décrit à [8, Bitstream generation], nous obtenons le fichier `.bin` à la fin de la synthèse du code. Ce fichier doit être accessible au système embarqué, soit par envoie par `scp` (cela nécessite une recompilation du buildroot), soit *via* un système de fichier NFS monté depuis l'Armadeus.

Après avoir chargé le module noyau (`fpgaloader.ko`) permettant le transfert du binaire dans le FPGA, l'implantation du composant s'obtient par :

```
dd if=ComposantVHDL.bin of=/dev/fpga/fpgaloader
```

3.1.2 Communication

Le projet Armadeus fournit un exemple de communication entre le processeur et le FPGA par cartographie (*mappage*) mémoire : `fpgaregs`. Cet exemple illustre la programmation sous GNU/Linux pour placer les valeurs appropriées sur les bus d'adresse et de données, utile pour débugguer rapidement les diverses versions de vos firmwares.

Par exemple dans le cas présent, l'exécution de

```
fpgaregs 02f4 1
```

permet d'allumer la diode connectée par l'anode au FPGA et par la cathode à la masse, tandis que

```
fpgaregs 02f4 0
```

met le bit de poids faible du bus de données à 0 et donc éteint la LED.

Depuis l'espace utilisateur, l'exploitation de cette cartographie de la mémoire pour communiquer avec le FPGA s'obtient en C par :

```
int MEM = open("/dev/mem", O_RDWR | O_SYNC);
mappage_FPGA = mmap(0, MAP_SIZE=4096, PROT_READ | PROT_WRITE,\
MAP_SHARED, MEM, FPGA_BASE_ADDR=0x12000000);
*(u16*) (mappage_FPGA + AddrxDI) = DataxD;  // ecriture
DataxD = *(u16*) (mappage_FPGA + AddrxDI);  // lecture
```

La même action s'obtient depuis un module noyau par :

```
mappage_FPGA = (void *) ioremap(FPGA_BASE_ADDR, 4096) ;
*(u16*) (mappage_FPGA + AddrxDI) = ; // ecriture
DataxD = *(u16*) (mappage_FPGA + AddrxDI); // lecture
```

Le lecteur averti aura remarqué que mise à part l'initialisation du mappage de la mémoire, les accès sont identiques, que l'on soit en espace utilisateur, ou noyau. On peut donc avoir une entête commune entre les deux codes résultants :

```
#define FPGA_BASE_ADDR 0x12000000
#define MAP_SIZE 4096
#define MAP_MASK ( MAP_SIZE - 1 )
void * mappage_FPGA;
void ecriture_fpga(unsigned short AddrxDI, unsigned short DataxD){
*(unsigned short*)(mappage_FPGA + AddrxDI) = DataxD;}
unsigned short lecture_fpga(unsigned short AddrxDI){
return *(unsigned int*)(mappage_FPGA + AddrxDI);}
```

On remarquera que la lecture nécessite une conversion de type (*cast*) vers entier non-signé `unsigned int`. Ceci est uniquement valable en espace utilisateur, et la raison nous est inconnue. Si la conversion de type est un mot de 16 bits (`unsigned short`), on ne récupère alors que les 8 bits de poids faible.

Il est intéressant d'estimer les temps de communication entre l'ARM et le FPGA, en fonction de ces 3 modes (communication depuis le shell, depuis un programme en C en espace utilisateur ou depuis l'espace noyau). Pour ce faire, une séquence commune va être réalisée, utilisant la lecture (RDxBI = '0' and CSxBI = '0') d'un registre (Compteur) qui sera incrémenté par les impulsions qu'il reçoit de l'horloge ClkxCI (96 MHz) fournie sur la broche 55 (L32P_4) du FPGA, lorsque notre LED sera allumé. Ce compteur peut par ailleurs être remis à zéro (cf compteur de fréquence, section 4).

La séquence de test suivante, permettra d'établir ces différents temps :

Action	Script sh	Appel C
RAZ<='1'	fpgaregs 02f0 1	ecriture_fpga(0x02f0,1) ;
RAZ<='0'	fpgaregs 02f0 0	ecriture_fpga(0x02f0,0) ;
LED<='1'	fpgaregs 02f4 1	ecriture_fpga(0x02f4,1) ;
LED<='0'	fpgaregs 02f4 0	ecriture_fpga(0x02f4,0) ;
Compteur(15 downto 0)	fpgaregs 0670	lecture_fpga(0x0670) ;
Compteur(31 downto 16)	fpgaregs 0674	lecture_fpga(0x0674) ;

Voici les résultats du script sh, suivi du C et du module noyau :

```
# ./Get.sh
read 0x0011 at 0x12000674
read 0x0497 at 0x12000670
```

Nous obtenons dans tous les cas de mesures d'abord le mot de poids fort suivi du mot de poids faible. Le calcul du nombre de cycles détectés par le compteur est obtenu par exemple sous GNU/Octave par `0x0011*65536+0x0497=1115287`. Ceci correspond au nombre d'oscillations de l'horloge de référence à 96 MHz fournie au FPGA par le signal ClkxCI (l'utilisation de ce signal comme référence de temps n'est pas judicieuse car il est issu d'une multiplication d'un oscillateur de référence à 32 kHz et présente des fluctuations de plus d'un kilohertz autour de sa valeur nominale. Cette précision suffit cependant pour cette démonstration).

Ces mêmes mesures sont faites pour le programme C compilé pour une exécution depuis l'espace utilisateur, et le module noyau et donnent respectivement :

```
# ./inter
Compteur=0
Compteur=56

# dmesg
...
Compteur=0
Compteur=27
```

Le temps d'éclairement de la LED est mesuré à l'oscilloscope à titre de comparaison avec la valeur du compteur :

Méthode	Compteur	Compteur/96MHz	Temps oscillo
sh	1115287	11,618 ms	11,62 ms
espace utilisateur	56	583 ns	585 ns
module noyau	26	270 ns	269 ns

Nous observons un ordre prévisible des performances, avec un interpréteur shell excessivement lent comparé à l'exécution en espace utilisateur d'un programme C, avec les performances optimales lors de l'accès au FPGA depuis le module noyau.

4 Application au compteur de fréquences

4.1 Principe du compteur réciproque

Un compteur conventionnel de fréquence f_i rapporte une valeur numérique C_i du nombre d'événements observables (transition d'un événement oscillant au delà d'un seuil) et discernables (nécessite des tensions suffisantes pour faire basculer les transistors, éléments de base du calcul numérique, et composant le FPGA) pendant un laps de temps t_g d'intégration défini, nommé temps de porte (*gate time*). On détermine alors la fréquence du signal oscillant par le rapport du registre $(C_i \pm 1)$ contenant la valeur du compteur et de $t_i + \Delta(t)$. Ce $\Delta(t)$ n'est pas constant dans le temps, et limite la précision d'un tel compteur (Fig. 3).

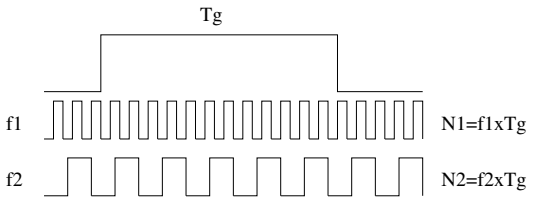


FIG. 3 – Principe de fonctionnement du compteur réciproque.

Le compteur réciproque permet de nous affranchir en grande partie de $\Delta(t)$, en mesurant le nombre d'oscillations C_{ref}, d'une seconde fréquence f_{ref} dite de référence. On obtient alors la fréquence inconnue par : $f_{ref} \times C_i/C_{ref}$. L'incertitude relative $(\Delta f/f)$ sur la valeur de la fréquence mesurée est alors de : $\frac{\Delta f}{f} = \frac{1}{t_g}\left(\frac{1}{f} + \frac{1}{f_{ref}}\right)$, en supposant l'incertitude sur les compteurs de $\Delta C_i = 1$.

4.2 Synthèse du compteur de fréquence en VHDL

Le compteur utilise un temps de porte (*gate time*) défini de façon logicielle [10, 11, 12]. Les N compteurs sont déclenchés sur le front montant du signal de porte et sont arrêtés par le front descendant. Ces N mesures sont alors transmises au processeur pour traitement (quotients pour revenir à une mesure relative à l'oscillateur de référence, s'affranchissant donc de l'incertitude sur le temps de porte).

Le code VHDL nécessite en général un fonctionnement synchrone. Mais pour le comptage des oscillations (+1 à chaque front montant <=> rising_edge), un fonctionnement asynchrone peut convenir.

```
signal Compteur : std_logic_vector (31 downto 0);
process ( Signal, GateTIME, RAZ )
begin
 if RAZ = '1' then -- Remise @ 0
Compteur <= "00000000000000000000000000000000";
 elsif rising_edge ( Signal )  and GateTIME = '1' then
Compteur <= Compteur + 1;
 end if ;
end process ;
```

La fréquence maximum de fonctionnement donnée lors de la synthèse par l'ISE de Xilinx est estimée à 194 MHz, mais des fréquences allant jusqu'a 250 MHz on pu être mesurées. Le signal qui nous intéresse, issu d'un oscillateur 433 MHz, sera donc divisé (Zarlink SP8402) et mis en forme avant d'arriver au FPGA.

4.3 Exploitation et performances du compteur

La réalisation d'un instrument n'est qu'une première étape, préliminaire à sa caractérisation et sa qualification dans les conditions d'utilisation qui vont nous intéresser. Nous allons donc dépasser les aspects purement informatiques de ces développements pour nous pencher sur la métrologie de la mesure de la fréquence et ainsi estimer les performances du compteur réciproque que nous avons réalisé. Le lecteur qui n'est pas concerné par l'application concrète de mesure de fréquence peut passer à la section suivante, bien qu'il nous semble utile d'insister sur le fait que la réalisation d'un instrument dépasse l'aspect purement informatique et électronique des développements.

Nous commençons par une utilisation démontrant le bon fonctionnement du montage (Fig. 5) avec une mesure de température observée comme variation de fréquence de l'oscillateur 433 MHz. des refroidissements et échauffements successifs sont clairement visibles sur la Fig. 6.

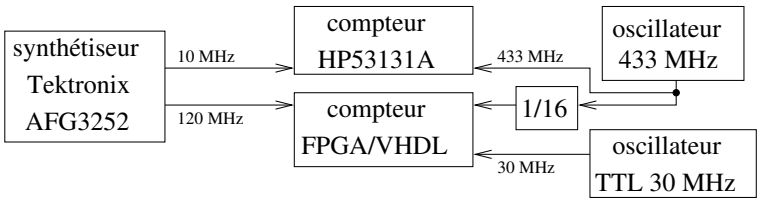


FIG. 4 – Schéma du montage visant à mesurer le bruit de mesure du compteur de fréquence. Un compteur de bonne qualité (Agilent 53131A) sert de référence par rapport à laquelle nous comparerons notre compteur réciproque. Le signal de référence de ces deux compteurs est synthétisé par un Tektronix AFG3252 (deux voies), d'une part pour former le 10 MHz de référence du compteur Agilent, et d'autre par 120 MHz pour notre compteur réciproque. Le signal de mesure qui nous intéresse, un oscillateur à 433 MHz sensible à la température, est divisé par 16 afin de rentrer dans la gamme de fréquences de fonctionnement du FPGA. Un oscillateur 30 MHz sert de référence secondaire et illustre la souplesse de la configuration du FPGA puisque nous pouvons rapidement étendre le compteur 2 voies à 3 voies.

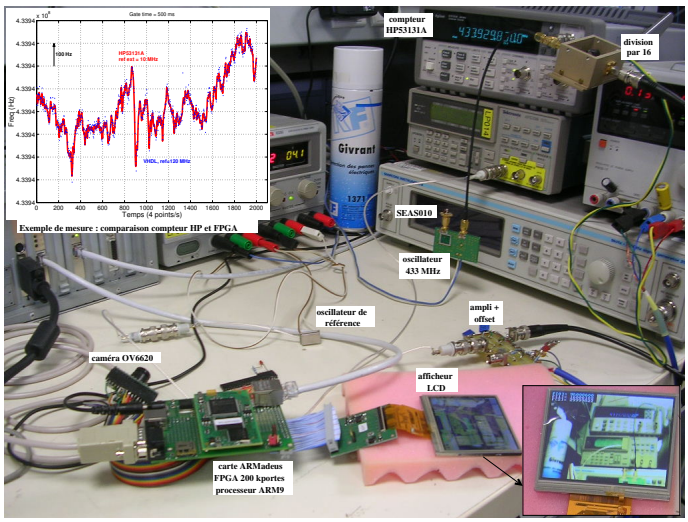


FIG. 5 – Prototype d'un instrument capable de compter deux fréquences autour de 433 MHz (divisée par 16 ≈ 27 MHz) relativement à une référence autour de 100 MHz, simultanément avec une capture d'image en 56 ms et un affichage des résultats de ces mesures sur écran LCD. La mesure de fréquence présentée en haut à gauche a été obtenue avec une temps de porte de 1 s (bleu), et est comparée avec la mesure d'un compteur Agilent 53131A (courbe rouge) : les deux courbes coïncident visuellement, mais une analyse quantitative est fournie dans le texte.

Une première analyse purement visuelle (Fig. 7 et courbe insérée dans Fig. 5) permet de comparer les performances de notre circuit avec celles d'un compteur de fréquence d'excellente qualité, le Agilent 53131A qui nous servira de référence au cours de cette discussion [13]. Il apparaît par exemple clairement que la tendance de la mesure est correcte, avec des fluctuations à long terme de la fréquence correctement observées, mais qu'avec un temps de porte de 100 ms notre mesure est plus bruitée que celle du compteur Agilent. Les bruit de mesure semble comparable entre les deux instruments pour un temps de porte de 1 s, qui sera désormais le paramètre sélectionné.

Au delà de cet aspect purement visuel, des outils ont été développés pour quantifier les performances d'oscillateurs. L'outil le plus couramment utilisé est la variance d'Allan $\sigma_y(\tau)$ [14], qui calcule l'écart type des mesures moyennées sur un intervalle de temps τ :

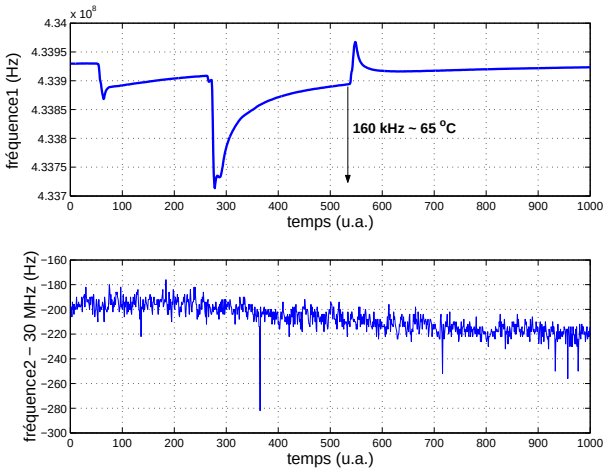


FIG. 6 – Mesure de température (courbe du haut) observée sous forme de dérive de fréquence d'un oscillateur 433 MHz spécialement conçu à cet effet. Le résonateur est refroidi à deux reprises avec une bombe givrante (dates 60 et 280) et chauffé à la date 550. La mesure est référencée sur la sortie 120 MHz du synthétiseur de fréquence Tektronix AFG3252. Un oscillateur TTL de 30 MHz en entrée d'un second compteur (courbe du bas) présente une stabilité de l'ordre de la dizaine de hertz relativement à cette référence.

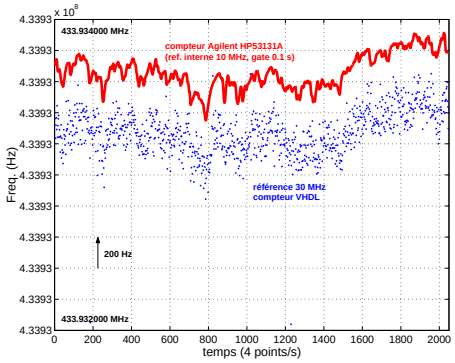


FIG. 7 – Exemple de mesure d'un oscillateur 433 MHz en prenant pour référence un oscillateur TTL à 30 MHz (courbe bleue). Comparaison avec la mesure simultanément obtenue avec un compteur Agilent 53131A programmé avec un temps de porte équivalent à celui utilisé par notre montage. Temps de porte de 100 ms (courbe rouge). Un offset à été volontairement ajouté pour mieux distinguer les deux courbes

$$\sigma_y(\tau) = \sqrt{\frac{1}{2(M-1)} \sum_{i=1}^{M-1} (y_{i+1} - y_i)^2}$$

pour M mesures de la série temporelle y régulièrement distribuées dans des segments de τ secondes.

$$y_n = \left\langle \frac{\delta f}{f} \right\rangle_n$$

pour une variation relative de fréquence δf autour de f.

Cet quantité $\sigma_y(\tau)$ estime donc les fluctuations de fréquence de l'oscillateur en fonction du temps d'intégration du compteur : le bruit d'un oscillateur n'est pas une quantitée intrinsèque mais dépend de la durée sur laquelle il est interrogé. À court terme, un oscillateur présente des fluctuations associées au bruit de l'électronique d'entretien de l'oscillation, auquel s'ajoute éventuellement un bruit de mesure (lié au compteur et non à l'oscillateur). À long terme, un oscillateur dérive du fait des fluctuations lentes de température (effets thermiques sur le résonateur ou l'amplificateur) ou de tension d'alimentation. Afin de comparer les divers oscillateurs que nous utilisons, nous considérerons toujours la variation *relative* de fréquence $\Delta f/f$, quantité sans unité qui permet donc de comparer les performances de systèmes fonctionnant à des fréquences f différentes.

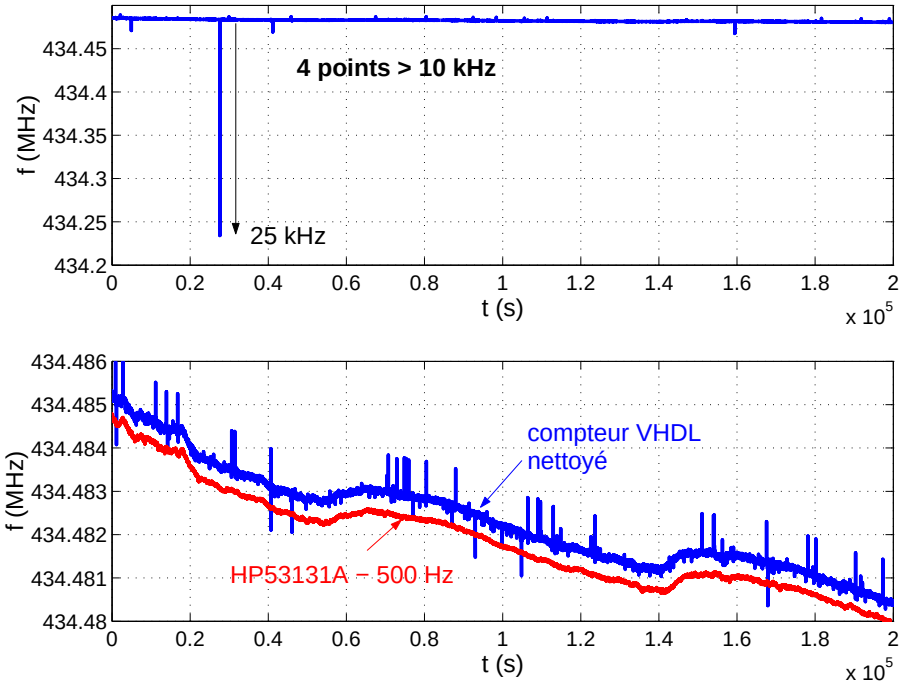


FIG. 8 – Évolution temporelle au cours d'un week end de la fréquence d'un oscillateur à 433 MHz comprenant un résonateur sensible à la température. En haut, mesures brutes comprenant 4 points présentant un écart de fréquence de plus de 10 kHz par rapport à leur voisin. En bas, en bleu la mesure obtenue par compteur dans le FPGA, nettoyée des points présentant un écart de plus de 1 kHz avec leur voisin (ce nettoyage a affecté moins de 30 points sur les 200000 acquis), et en rouge la mesure obtenue simultanément sur compteur Agilent 53131A, translatée de 500 Hz par soucis de clarté.

Une acquisition longue (3 jours) avec une fréquence de 1 s permet de mettre en évidence diverses propriétés de l'oscillateur. L'évolution temporelle du signal (Fig. 8) montre d'une part une dérive long terme du signal – probablement associé à une dérive thermique ou de la tension d'alimentation de l'oscillateur – observé à la fois par notre compteur et le compteur Agilent. Par ailleurs, notre compteur présente quelque mesures erronées, aisément éliminées par post-traitement numérique mais sur lesquelles nous reviendrons plus tard.

La variance d'Allan calculée sur cet ensemble de points est illustrée par les ronds rouges sur la Fig. 9 pour le compteur synthétisé dans le FPGA et le compteur Agilent. Nous constatons que les deux calculs fournissent le même résultat à long terme ($\tau > 700$ s) mais les deux courbes diffèrent pour des temps d'intégration plus courts.

Il est surprenant de constater que notre compteur, configuré pour mesurer simultanément un oscillateur 433 MHz et un oscillateur TTL 30 MHz, présente le même bruit de fréquence pour

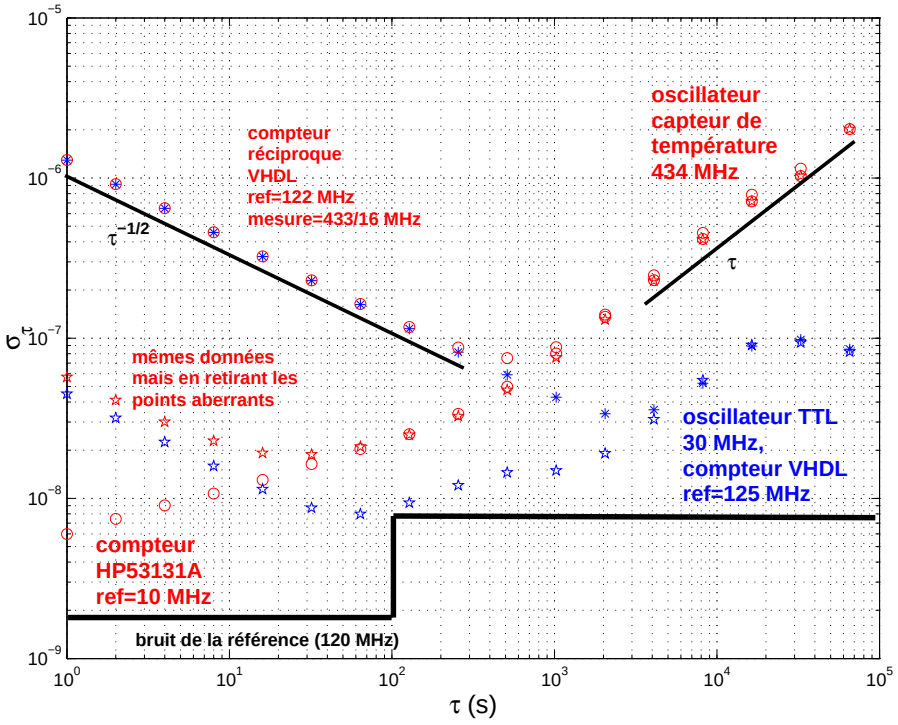


FIG. 9 – Analyse quantitative de la stabilité de l'oscillateur par calcul de la variance d'Allan. Le temps de porte est de l'ordre de 1 s. La pente de la variance d'Allan en $\tau^{-1/2}$ correspond à une fluctuation en bruit blanc de fréquence. Le pente de la variance d'Allan en τ correspond à la dérive thermique de l'oscillateur. Les ronds rouges et les "*" bleus sont obtenus par traitement des données brutes, incluant quelque point aberrants présentant un écart de plus de 1 kHz par rapport à leurs voisins : la variance d'Allan à court terme est dominée par ces points aberrants. Les étoiles rouges et bleues sont obtenues par post-traitement numérique des signaux (retrait des points aberrants, affectant moins de 1 point sur 5000 de la séquence acquise), et se rapproches du signal "idéal" obtenu par compteur 53131A. Ce compteur professionel élimine par moyenne glissante le bruit blanc de fréquence de pente $\tau^{-1/2}$, d'où l'écart entre nos mesures et celles de ce compteur à $\tau < 40$ s.

ces deux oscillateur pour $\tau < 700$ s. La source de référence à 120 MHz a alors été qualifiée indépendamment et a démontré une bonne stabilité avec des fluctuations relatives de fréquences inférieures à 10^{-8} (trait noir en bas du graphique 9). Nous avons identifié la cause de cette variance élevée à court terme en éliminant les points aberrants de mesure aisément visibles sur la Fig 8 (haut). Seuls 4 points aberrants avec un saut de fréquence de plus de 10 kHz par rapport à leurs voisins sont la cause du seuil élevé de la variance d'Allan à court terme. En éliminant les points donc l'écart avec leur voisin est de plus de 1 kHz, le calcul de la variance d'Allan sur l'oscillateur 433 MHz et l'oscillateur TTL 30 MHz donne les courbes en étoile sur la Fig. 9.

Nous constatons que le résultat à long terme ($\tau > 700$ s) n'a pas été affecté, mais maintenant les deux mesures sont bien distinctes et suivent le résultat obtenu avec compteur Agilent jusqu'à $\tau = 30$ s. La correction n'a porté que sur 16 points parmi les 200000 acquis, mais cette extrême sensibilité de la variance d'Allan à très peu de point aberrants est un problème connu qui mériterait à être corrigé lors de la phase d'acquisition et de traitements des valeurs des compteurs.

Nous attribuons les points erronés à du bruit lors des mesures puisque le montage n'a pas été blindé et aucun soin particulier sur le filtrage des alimentations n'a été amené. Il serait souhaitable de réaliser un circuit convenablement agencé afin de blinder les divers signaux radiofréquences et

les isoler du bruit numérique amené par la caméra connectée au même circuit, sujet de la section qui suit.

La synthèse d'un grand nombre de compteurs actifs simultanément ($N >$ supérieur à quelques unités) n'a probablement pas de grand intérêt pratique compte tenu du nombre d'oscillateurs que nous sommes susceptibles d'interroger. Cependant, les perspectives de traitement plus complexes des données des compteurs en implémentant des fenêtres pondérées et des moyennes glissantes [15] – opération nécessitant plus d'un registre par compteur – sont ouvertes puisque nous n'occupons aujourd'hui qu'une fraction réduite (≈10%) de la surface du FPGA.

5 Interfaçage d'une caméra CMOS

Ayant acquis la maîtrise de l'incrément d'un compteur sur un front de signal radiofréquence et du transfert vers le microprocesseur de la valeur accumulée sur le FPGA, nous désirons maintenant aborder un problème plus complexe. Un capteur optique CMOS est un composant numérique, qui fournit à chaque front d'un signal d'horloge de quelques dizaines de MHz la nouvelle valeur d'un pixel. Au lieu de simplement incrémenter un compteur, nous allons stocker dans une mémoire tampon les valeurs de ces pixels afin de restituer une image au processeur. L'objectif d'utiliser un FPGA comme coprocesseur dédié à cette tâche est de limiter le temps d'occupation du processeur pour l'acquisition d'images par une solicitation uniquement lorsque la RAM synthétisée dans le FPGA est pleine.

5.1 Le capteur OV6620

Le capteur OV6620 a été acquis auprès de Lextronic [16], monté sur un circuit imprimé et équipé d'une optique digne d'une webcam, sous la nomenclature CA88. D'une résolution de 100 kpixels, l'OV6620 n'est évidemment pas concurrent des appareils photographiques numériques récents, mais suffisant pour se familiariser avec les concepts de base de l'acquisition d'images et une restitution sur écran LCD de résolution comparable. Noter qu'il faut absolument éviter d'acquérir la version 3 Mpixels (OV3620) de ce capteur qui, sous une apparence de signaux de contrôle et de méthodes d'acquisition d'images similaires, *ne fonctionne pas avec la configuration par défaut fournie par le constructeur*, sachant que ledit constructeur refuse de fournir les valeurs des registres de configuration permettant l'utilisation du capteur sans le paiement d'un forfait dépassant largement les moyens du développeur amateur.

À titre indicatif, nous fournissons pour le lecteur désireux d'exploiter un capteur CMOS Omnivision OV3620 (3 Mpixels, adresse I^{2}C 0x60)) la séquence des registres (première colonne) et des valeurs à y placer (seconde colonne). Noter que la majorité de ces registres ne sont pas documentés dans la datasheet et que leur redéfinition à plusieurs reprises lors de l'initialisation fait penser à une série d'opcodes plutôt que des valeurs de registres de configuration

12 80	36 4C	36 4C	0F 42	43 00	34 58	32 36
12 80	39 F3	39 F3	14 C6	45 80	12 00	03 40
E5 3E	13 0D	13 C5	15 10	48 C0	17 10	02 86
E4 26	24 58	24 58	33 09	49 19	18 90	2D 00
E1 67	25 48	25 48	34 50	4B 80	19 01	00 00
F8 00	12 80	12 80	36 00	4D C4	1A C1	01 90
23 00	12 80	12 80	37 04	35 4C	32 36	12 00
F8 01	E5 3E	12 80	38 52	3D 00	03 40	10 43
FF B0	F8 01	12 80	3A 00	3E 00	11 02	13 C7
12 80	12 80	13 C7	3C 1F	3B 18	12 20	15 12
0E 05	0E 05	09 00	44 00	33 19	17 10	
12 04	12 04	0C 08	40 00	34 5A	18 90	
3C 0C	3C 0C	0D A1	41 00	3B 08	19 01	
33 37	33 37	0E 70	42 00	33 09	1A C1	

Nous avions déjà décrit ce capteur dans le cadre de son interfaçage avec un processeur de la série Coldfire. Cette fois, au lieu de ralentir la vitesse d'acquisition des images afin de permettre au processeur de capturer chaque valeur de pixel sur son bus de données, nous allons nous servir du FPGA pour synthétiser une RAM qui générera une interruption pour prévenir le processeur qu'elle est pleine. De cette façon, nous sommes capables d'utiliser le capteur CMOS avec une capture d'image (registre 0x11 programmé par I2C à 0x02) en 56 ms, fréquence optimale de fonctionnement de la caméra utilisée afin d'éviter d'obtenir une image floue lorsque le capteur est en mouvement, tout en étant compatible avec la vitesse du FPGA et sa communication avec le processeur. Cette méthode limite le temps d'occupation du processeur puisque celui-ci n'est solicité que chaque fois que le tampon du FPGA est plein.

5.2 Configuration de la caméra : utilisation du bus I^{2}C

Le support I2C est compilé en statique dans le noyau fourni par défaut avec la carte Armadeus. La communication se fait avec le device `/dev/i2c-0`. La configuration de la caméra ne nécessite que des phases d'écriture afin de configurer les registres de la caméra qui définissent le protocole de communication et la vitesse de rafraîchissement des images (et donc la vitesse de transfert des pixels).

```
int write_byte(int fd, unsigned char reg, unsigned char value)
{
    unsigned char buf[2] = {reg,value}; // initialise a data buffer with
                                        // address and data

    // create an I2C write message
    struct i2c_msg msg = { CA88=0x60, 0, sizeof(buf), buf };
    // create a I2C IOCTL request
    struct i2c_rdwr_ioctl_data rdwr = { &msg, 1 };


    printf("write ref %02X value %02X\n",reg, value);
```

```
    if ( ioctl( fd, I2C_RDWR, &rdwr ) < 0 ){
printf("Write error\n");return -1;}
    return 0;}
```

Voici les registres modifiés, et leurs valeurs respectives [17] :

```
//Mode RGB, Auto (Gain Control & White Balance)
write_byte (i2c, 0x12, 0xa4);
write_byte (i2c, 0x11, 0x02); // 1 img / 56 ms
write_byte (i2c, 0x12, 0x2c);
write_byte (i2c, 0x13, 0x21); // Format 8 bit
```

L'architecture de la mesure est différente du cas du compteur : dans ce premier cas, le processeur imposait la cadence en définissant le temps de porte et en lisant les résultats de la mesure après le temps d'acquisition. Cette fois, la caméra nous impose la cadence en fournissant les signaux PCLK (nouveau pixel sur le bus de données), HREF (nouvelle ligne) et VSYNC (nouvelle image). Nous devons donc synthétiser dans le FPGA la logique associée à ces trois signaux : attendre une nouvelle image (condition sur VSYNC), et tant que les pixels de chaque ligne sont valides (HSYNC haut), nous allons lire les données fournies par le capteur CMOS. Ces données sont accumulées dans une RAM servant de tampon afin de libérer la charge du processeur.

Le signal VSYNC informe sur le début de l'image (coin haut à gauche). Ensuite chaque pixel est envoyé séquentiellement, et est valide si Href et Pclk sont à l'état haut. La solution retenue consiste à incrémenter une adresse de registre à chaque front descendant de Pclk :

```
process(Vsync, Href, Pclk, Nb_Img) is
begin
  if Vsync='1' then
     ADDRA<="00000000000";
   else if Href='1' and falling_edge(Pclk) then --and Nb_Img=1 then
           ADDRA<=ADDRA+4;
        end if;
   end if;
end process;
```

ADDRA est incrémenté par pas de 4, du fait que nous n'avons pas maîtrisé l'implémentation de la RAM et la communication entre FPGA et processeur ARM9. Cette valeur a été validée de façon expérimentale, et nous l'avons conservée. Il serait cependant important d'en comprendre la raison fondamentale, et notamment l'organisation de la valeur 32 bits reçue lors de l'interrogation des registres synthétisés dans le FPGA (http://www.armadeus.com/wiki/index.php?title=FPGA_register). Ces points sont en cours de validation par les membres de l'association Armadeus et une version corrigée pour des lectures 16 ou 32 bits de `fpgaregs` et la RAM associée dans le FPGA `bBRAMTest` devrait être disponibles à la date de parution de cet article.

5.3 Acquisition des images

L'adresse ainsi obtenue est utilisée pour le contrôle du port d'entrée-sortie 8 bit d'une RAM qui comporte un second bus de 16 bits.

```
   -- RAMB16_S9_S18: Virtex-II/II-Pro, Spartan-3/3E 2k/1k x
8/16 + 1/2 Parity bits Parity bits Dual-Port RAM
   -- Xilinx HDL Language Template, version 9.2i
```

Cette RAM est un des nombreux composants VHDL génériques fourni par Xilinx. Leur implémentation est relativement simple à mettre en œuvre. Ici, le premier port (8 bit) est destiné à recevoir les valeurs des pixels envoyés par la caméra, et n'est donc considéré que comme une entrée. Le second bus de 16 bits convient parfaitement au bus de données existant entre l'ARM et le FPGA, et il

n'est donc utilisé qu'en sortie pour communiquer au monde extérieur les informations que la RAM contient.

L'inconvénient majeur de cette solution est que la RAM synthétisable ne permet pas de stocker une image entière. Il faut donc faire transiter en plusieurs petits paquets les informations contenues dans le FPGA au noyau Linux. Pour ce faire, la ligne `TIM1`, associée à une interruption matérielle, reliant le FPGA à l'ARM va être utilisée. Elle permet de générer une interruption lors d'une transition haut/bas, afin de signaler un évènement au noyau. Un module doit s'être déclaré comme capable de gérer cet évènement : la fonction d'interruption `f_interruption()` est initialisée à l'insertion du module par `insmod` :

```
request_irq(IRQ_GPIOA(1), f_interruption, SA_INTERRUPT, "TIM1", NULL);
set_irq_type( IRQ_GPIOA(1), IRQF_TRIGGER_FALLING );
```

Cette interruption est générée lorsque notre codeur d'adresse `ADDRA` dépasse une valeur que l'on ajuste expérimentalement afin que le temps de lecture corresponde exactement au temps qu'il faut à la caméra pour finir de remplir la mémoire.

```
process(ADDRA, Vsync, Nb_Img) is
begin
    if Nb_Img=1 and (ADDRA>"10111110000" or (Vsync='1'  and ADDRA>"0")) then
       TIM1<='0';
    else
       TIM1<='1';
    end if;
end process;
```

Le noyau Linux ayant été prévenu que la RAM va bientôt être pleine, la lecture est amorcée. Une solution basée sur l'utilisation de 2 RAMs individuelles – l'une en cours de remplissage pendant que l'autre est vidée – a échoué faute de temps (et de compétences dans le domaine), bien que des résultats préliminaires montrent que cette solution est réalisable (et souhaitable).

La lecture des données dans la RAM se fait par la fonction suivante :

```
unsigned short lecture_fpga_ram(unsigned short addr){
  static unsigned short resultat;
  static unsigned long resl;
  resl= *(unsigned long*)(FPGA_Addr+addr);
  resultat = ((resl&0xff0000)>>8)+(resl&0x0000ff);
  return resultat;
}
```

Cette fonction traduit du côté Linux l'architecture de la RAM dans le FPGA : la donnée lue (sur 32 bits) est retouchée pour former un mot de 16 bits représentatif de la valeur du pixel stocké en RAM. L'adresse `addr` est incémentée de 4 entre deux appels à cette fonction afin de lire deux mots successifs en RAM :

```
static int f_interruption(int irq, void *dev_id, struct pt_regs *regs)
{static int i;
 static unsigned short ushortsize;
 for(i=0;i<256;i++){
    ushortsize=lecture_fpga_ram(i*4);
    img1[Indice_Tableau]=ushortsize;  // unsigned short *img1; recupere 2 pixels
    Indice_Tableau++;
 }
 return(Indice_Tableau);
}
```

À la fin des interruptions (plus précisément à la fin d'un temps pré-défini `usleep()` dans notre implémentation actuelle), le programme utilisateur récupère le tableau `img1` par l'intermédiaire d'un appel à la fonction `read()`, associée dans le noyau à la fonction `read_cmos()`. Il serait probablement souhaitable d'utiliser un signal du type `SIGUSR` afin de réveiller le programme en espace utilisateur à la fin d'une acquisition d'image au lieu d'une attente d'une durée prédéfinie.

La fonction `read()` du C étant définie comme bloquante, la façon "normale" de gérer le retour d'information serait de ne renvoyer l'image que si le nombre prévu d'octets a été effectivement lu depuis la caméra. Cela nécessite d'implémenter quelques sécurités additionnelles pour valider l'image acquise (décompte du nombre de lignes et de pixels par exemple). Cette méthode n'a pas été proposée ici car elle suppose que les images acquises sont parfaites, cas rarement vérifié en phase de debuggage de la couche matérielle.

```
static ssize_t read_cmos(struct file *file, char *buf, size_t count, loff_t *ppos)
{copy_to_user(buf, (unsigned char *)img1 , count);}
```

Du point de vue de l'espace utilisateur, la requête pour une nouvelle image est initialisée par un `ioctl()` qui demande au module noyau de déclencher une capture sur le FPGA. Suite à une attente de plus de 56 ms, l'espace utilisateur effectue un appel à `read()` pour récupérer le nombre attendu de pixels. Le résultat d'une acquisition, après traitement pour convertir le format de Bayer en image couleur RGB (section 5.4), est présenté sur la Fig. 10.

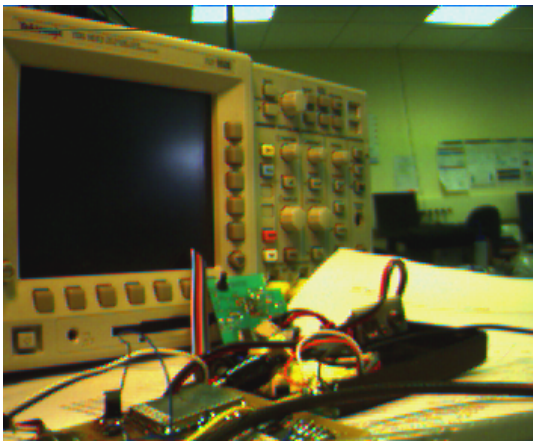


FIG. 10 – Exemple d'image capturée en pleine vitesse (pas de ralentissement de l'horloge du capteur CMOS OV6620) par le FPGA, transmise au processeur pour traitement en vue d'une conversion en image couleur et affichage sur écran LCD puis diffusion par réseau ethernet.

5.4 Affichage sur écran LCD

L'affichage sur framebuffer est particulièrement simple puisqu'il s'agit d'une zone mémoire dont le contenu est directement transféré sur l'écran (l'équivalent moderne du vénérable mode 13 des cartes VGA [19]). Ainsi, pour un écran de $long \times large$ pixels, la zone mémoire $(x + y \times large) \times N$ contient l'information de couleur du pixel de coordonnées (x, y) si N octet(s) sont utilisés pour chaque pixel.

Avant d'afficher des pixels sur le LCD, il faut l'initialiser (cf [18]) :

```
int fbfd = 0;
struct fb_var_screeninfo vinfo;
struct fb_fix_screeninfo finfo;
long int screensize = 0;
```

```
char *fbp = 0;
init_lcd(){
fbfd = open("/dev/fb0", O_RDWR);
ioctl(fbfd, FBIOGET_FSCREENINFO, &finfo)
ioctl(fbfd, FBIOGET_VSCREENINFO, &vinfo)
// Figure out the size of the screen in bytes
screensize = vinfo.xres * vinfo.yres * vinfo.bits_per_pixel / 8;
// Map the device to memory
fbp = (char *)mmap(0, screensize, PROT_READ | PROT_WRITE, MAP_SHARED, fbfd, 0);
}
```

Il est alors possible de spécifier la couleur R,G,B du pixel (x,y) par la fonction `pix()` :

```
void pix(int x, int y, int r, int g, int b){
long int location = 0;
location = (x+vinfo.xoffset) * (vinfo.bits_per_pixel/8) +
           (y+vinfo.yoffset) * finfo.line_length;
unsigned short int t = ((r&0xf8)>>3)<<11 | ((g&0xfc)>>2) << 5 | ((b&0xf8)>>3);
*((unsigned short int*)(fbp + location)) = t;
}
```

Dans notre cas, les valeurs des pixels, envoyées par la caméra, suit la séquence suivante : rgbgrgbgrgbgrgbgrgb ... Sachant que la caméra possède un filtrage de Bayer, quatre pixels réels (rg_1bg_2), correspondent à deux pixels image (rg_1b, rg_2b).

Afficher notre image conformément aux spécifications de ce filtre devient (Fig. 11) :

```
for(j=0;j<240;j++){//Hauteur du LCD
  for(i=0;i<160;i++){//Demi-Largeur du LCD
    pix(2*i, j, Mon_Image[j*704+4*i+2], Mon_Image[j*704+4*i+1], Mon_Image[j*704+4*i]);
    pix(2*i+1, j, Mon_Image[j*704+4*i+2], Mon_Image[j*704+4*i+3], Mon_Image[j*704+4*i]);
  }
}
```

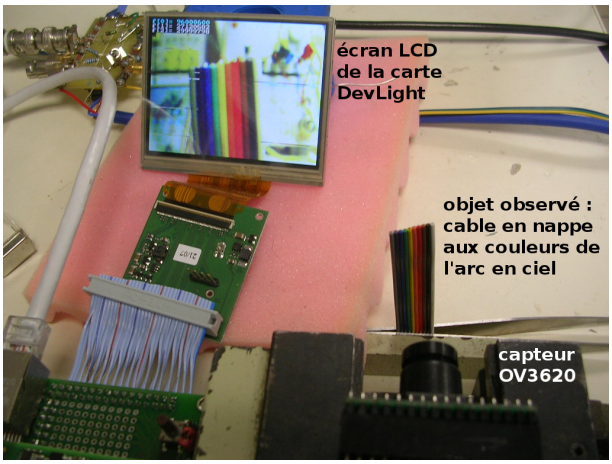


FIG. 11 – Affichage d'une image sur l'écran LCD fourni avec la carte d'évaluation DevLight : l'image est acquise *via* le FPGA sur un capteur optique OV6620, transmise en espace noyau Linux pour répondre à la requêtre d'un programme en espace utilisateur. Ces données sont converties du format Bayer en RGB pour être affichées *via* le framebuffer `/dev/fb0`.

6 Conclusion et perspectives

Nous avons présenté l'utilisation d'une plateforme comprenant un processeur généraliste sur lequel est exécuté un système complet GNU/Linux, couplé à une matrice de portes logiques reconfigurables (FPGA) pour la réalisation d'un compteur réciproque fonctionnant dans la gamme radiofréquence (<200 MHz), et pour la capture à près de 20 images/secondes d'images issues d'un capteur optique CMOS. Nous avons pris soin de qualifier ces montages et d'en optimiser les performances en tentant de placer les opérations nécessitant une réaction rapide dans la logique reprogrammable, afin de limiter la surcharge de travail du processeur sur lequel tourne le système d'exploitation multitâches.

Bien que ce travail réponde à nos attentes et fournisse un résultat exploitable pour l'interrogation de capteurs basés sur la mesure de fréquence d'oscillateurs dérivant avec la grandeur physique mesurée, un certain nombre de points restent à éclaircir ou pourraient être améliorés :

- le compteur de fréquence présente des mesures grossièrement erronées que nous pouvons filtrer numériquement après acquisition, dont la source reste à identifier
- des points d'ombre subsistent sur la communication entre une RAM implémentée dans le FPGA et l'ARM9 : nous avons dû lire des valeurs sur 32 bits (deux cycles d'accès mémoire) quand seuls 16 bits nous intéressaient. Nous n'avons pas été capables de générer 32 bits pertinents ou de faire des requêtes en mémoire sur 16 bits (1 cycle)
- la lecture du tampon en RAM synthétisée dans le FPGA est déclenchée par interruption depuis le FPGA afin que le temps de lecture des données par le processeur soit exactement égal au temps de remplissage des derniers mots de la RAM par l'image. Cette solution, qui serait acceptable sur un système d'exploitation temps réel avec des latences bornées, n'est pas acceptable sous GNU/Linux. Il serait donc judicieux de faire fonctionner une solution de deux tampons appelés alternativement, l'un étant rempli par l'image issue du capteur CMOS pendant que les données de l'autre sont transférées au processeur,
- une image erronée peut être transmise au processeur faute de validation sur le nombre de lignes ou de pixels lus. Par ailleurs, nous n'avons pas exploité une fonctionnalité de notre capteur CMOS de rafraîchir plus rapidement un sous ensemble de l'image (ROI, *Region of Interest*) qui pourrait présenter un intérêt, notamment dans le couplage des capteurs radiofréquence avec une mesure optique pour la mesure de vibration de poutres en vibration : ces améliorations pourront être implémentées en cas de besoin,
- au cours de nos différents développements, nous avons tenté d'utiliser les avantages du code VHDL pour rendre générique (fonction VHDL `generate`) nos modules de comptage. Cependant, l'ISE de Xilinx génère une erreur : `"INTERNAL_ERROR :Xst :cmain.c :3111 :1.8.6.1"`. Le site web Xilinx signale qu'il s'agit d'une erreur connue en cours de correction pour les versions ultérieures (> 9.2i).

7 Remerciements

Ce travail a nécessité des interaction diverses avec des interlocuteurs expérimentés dans leur domaine d'expertise.

Nous remercions Julien Boibessot (Armadeus) pour son assistance et le temps qu'il a passé à refaire fonctionner notre carte Armadeus après quelques manipulations malheureuses (ne jamais essayer de flasher un fichier binaire destiné au FPGA à la place du bootloader), ainsi que l'ensemble des membres du chan `#armadeus` sur IRC (`irc.rezosup.org`).

Sébastien Euphrasie (FEMTO-ST) nous a assisté sur le développement en VHDL. Les mesures radiofréquences et l'analyse des données issues du compteur n'auraient pas été possibles sans l'aide de nos collègues de l'équipe Temps-Fréquence de l'institut FEMTO-ST : Pierre-Yves Bourgeois, Yann Kersalé, Gilles Martin et Enrico Rubiola ont contribué par leur expérience au succès de ce travail.

Références

[1] Un exemple au hasard : http://www.imec.be/ovinter/static_research/reconfigurable.shtml

[2] L. Cristaldi, A. Ferrero & V. Piuri, *Programmable instruments, virtual instruments, and distributed measurement systems : what is really useful, innovative and technically sound ?*, IEEE Instrumentation & Measurement Magazine (Sept. 1999), 20-27, met l'accent sur l'utilisation de processeurs dédiés au traitement du signal (DSP) plutôt que sur les FPGA, et les interfaces utilisateur.

[3] A. Ben Atitallah & P. Kadionik, *Linux et les systèmes embarqués multimédias*, **97** (Sept 2007), 75-81

[4] J. Daněček, F. Drápal, A. Pluháček, Z. Salčič, M. Servít, *DOP – a simple processor for custom computing machines*, J. of Microcomputer Applications (1994), **17**, 239-253

[5] Z. Salcic, *PROTOS – a microcontroller/FPGA-based prototyping system for embedded applications*, Microprocessors and microsystems **21** (1997), 249-256

[6] J. Boibessot, *Linux Embarqué pour tous !*, GNU/Linux Magazine France **92** (Mars 2007), 70-79, et le site web http://www.armadeus.org

[7] http://www.arm.com/products/CPUs/families/ARM9Family.html

[8] http://www.armadeus.com/wiki/index.php?title=FPGA

[9] http://www.xilinx.com/support/documentation/data_sheets/ds099.pdf

[10] J. Kalisz, R. Szplet, J. Pasierbinski & A. Poniecki, *Field-Programmable-Gate-Array-Based Time-to-Digital Converter with 200-ps Resolution*, IEEE Transactions on instrumentation and measurement **46** (1), (1997), 51-55

[11] R. Szymanowskia & J. Kalisz, *Field programmable gate array time counter with two-stage interpolation*, Rev. Sci. Instrum. **76** (2005), 045104

[12] J. Kalisz, *Review of methods for time interval measurements with picosecond resolution*, Metrologia **41** (2004), 17-32

[13] S.T. Dawkins, J.J. McFerran & N. Luiten, *Considerations on the Measurement of the Stability of Oscillators with Frequency Counters*, comptes rendus de la conférence IFCS-EFTF (2007), 759-764

[14] E. Rubiola, *The Leeson Effect* disponible à http://www.femto-st.fr/~rubiola/slides/leeson-effect-slides2006.pdf, p. 15

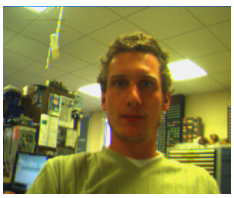
[15] E. Rubiola, *On the measurement of frequency and its sample variance with high-resolution counters*, Rev. Sci. Instrum. **77** (2005), 054703

[16] http://lextronic.fr/P1729-camera-numerique-ca88.html

[17] J.-M Friedt, S. Guinot, *Introduction au Coldfire 5282*, GNU/Linux Magazine France, **75** (Septembre 2005)

[18] http://thunder.prohosting.com/~bricks/linux/program/framebuffer.html

[19] A. LaMothe, *Teach Yourself Game Programming In 21 Days*, SAMS Publishing (1994), ou http://www.brackeen.com/vga/index.html



T. Rétornaz complète son Master 3I (Image, Informatique, Ingénierie) entre Besançon et Dijon. Il est membre des associations Projet Aurore et Sequanux (www.sequanux.org) pour la diffusion de la culture scientifique et du logiciel libre en Franche-Comté. Cette photo a été acquise avec la caméra présentée dans cet article.

J.-M Friedt est ingénieur dans la société Senseor (www.senseor.com), hébergé par l'institut FEMTO-ST de Besançon, et membre de l'association Projet Aurore (http://projetaurore.assos.univ-fcomte.fr/).