

# Découvrir le composant Zynq : processeurs et FPGA sur une même puce

## 1 Introduction

**Présentation du couple CPU+FPGA : partant de Armadeus, plateformes proposées par Xilinx. Disponibilité et coût des plateformes à base de Zynq. Adéquation pour les applications de SDR (redpitaya?).**

La séquence des tâches que nous nous proposons de suivre consiste en

1. programmation du CPU seul en l'absence de système d'exploitation
2. programmation du FPGA et communication avec le CPU en l'absence de système d'exploitation
3. installation de GNU/Linux sur le CPU
4. interfaçage du FPGA avec le CPU sous GNU/Linux au travers de l'espace utilisateur
5. interfaçage du FPGA avec le CPU sous GNU/Linux au travers d'un module noyau.

Ce document a un simple objectif : faciliter la prise en main de la plaquette Zedboard et éviter aux futurs utilisateurs de rencontrer les problèmes que j'ai pu rencontrer ces derniers temps. Je compléterai cet article au fur et à mesure de l'avancement de mes travaux sur la platine, aussi il se peut que vous le trouviez lors de votre consultation, comme succinct ou manquant de détails sur certains aspects. Le but étant avant tout d'éclaircir le sujet en dépit des documentations qui peuvent parfois faire défaut ou être complexes, et d'être le plus explicite possible. L'environnement de travail est basé sur une distribution *Debian* de GNU/Linux et cette suite de tutoriels sera parfaitement adaptée si vous êtes sous *Linux*.

## 2 Architecture de la carte

La plateforme de développement Zedboard est composée d'un composant Zynq incluant sur une même puce deux processeurs ARM et d'un FPGA de la famille Zc7020 :

- la partie CPU que l'on nommera PS pour *Processing System* est obligatoire, quelque soit l'application finale,
- la partie FPGA que l'on nommera PL pour Programmable Logic est facultative, pour l'implémenter il nous faudra décrire et spécifier un système matériel au travers des outils Xilinx.

La carte dispose de 54 entrées/sorties (que l'on nommera IO) qui sont interfacées au CPU au travers d'un multiplexeur nommé MIO : **Multiplexed Input Output**. C'est le cas par exemple des deux BTN8 et BTN9 qui sont respectivement les MI051 et MI050 situés au dessus des 8 switches. Il est également possible d'étendre le nombre de GPIO via un ou plusieurs bloc que l'on nomme EMIO *Extended MIO*, ce qui permettra essentiellement de récupérer des signaux provenant du FPGA dans le CPU sans passer par le bus de communication. La plupart des entrées/sorties de la carte sont routées depuis sur le FPGA, tel que nous le montre le schéma en figure 1.

Par exemple, les pins SCK, MISO, MOSI, CS0 du SPI1 sont uniquement accessibles depuis le CPU, ils sont reliés physiquement au PMOD A. On pourra cependant spécifier un second port SPI2 dont les signaux seront répercutables vers le FPGA directement. Le schéma le plus pertinent dont nous disposons à ce sujet est représenté en figure 2.

Comme vous pouvez le voir, seuls quelques périphériques bien précis pourront être routés vers le FPGA. L'interface entre la partie PS et la partie PL se fait au travers du bus AXi dont la documentation complète est référencée en bas de page. Nos interfaces AXi seront créées au travers des outils Xilinx et selon les besoins de notre application. Enfin la grande partie des IO de la carte sont routées sur le FPGA, c'est le cas notamment du codec AUDIO, de l'HDMI, des 5BTNS disposés en croix, des 8 Switches et des 8 LEDS, du connecteur FMC etc...

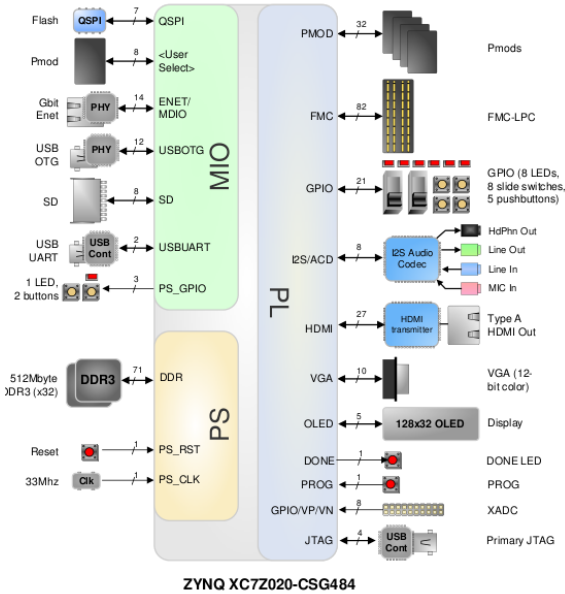


FIGURE 1 – Schéma bloc représentant les périphériques disponible sur la carte zboard.

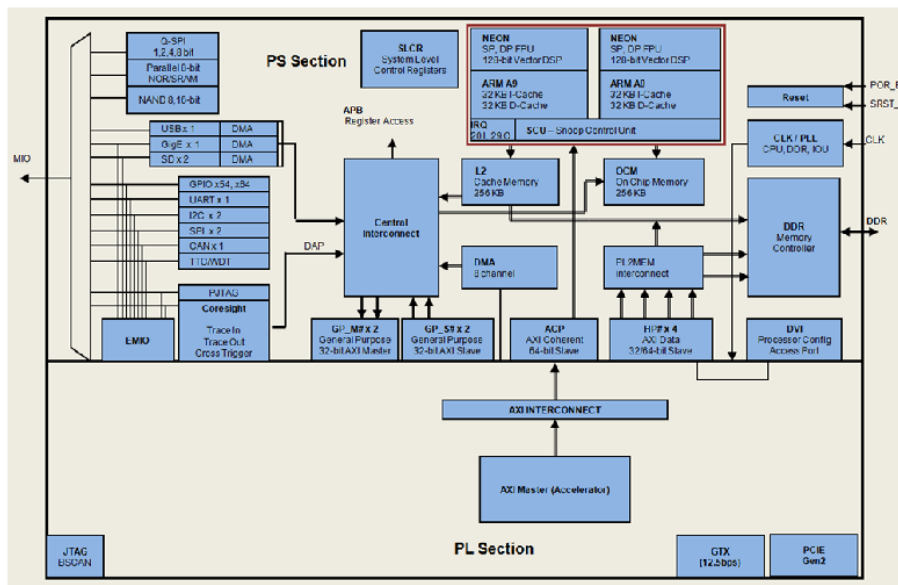


FIGURE 2 – Schéma bloc représentant les accès au périphérique depuis le PL et PS.

### 3 Première prise en main : outils de travail

#### 3.1 Outils de développements logiciels

Pour développer nos applications il nous faudra au minimum un compilateur de code C/C++ qui définisse la séquence de tâches à exécuter par CPU. Si l'on se cantonne aux outils Xilinx pour le moment, cette partie est gérée par l'outil SDK (*Software Development Kit*) basé sur Eclipse. En plus de faire office de compilateur, cet outil G va permettre de transférer notre code via la liaison JTAG virtuelle, mais aussi flasher un descriptif de comportement du FPGA (fichier .bit) si notre application y fait appel.

Si c'est le cas, le .bit sera généré grâce aux différentes étapes effectuées en amont, qui consisteront à

spécifier les interfaces matérielles telles que : la présence ou non d'une interface AXi, l'implémentation d'un bloc venant s'enficher sur le bloc *AXi interconnect* (par exemple un GPIO dans le cas d'une écriture vers les LEDs du FPGA, ou d'une lecture des BTNS disposés en croix, d'un bloc personnel spécifique). L'implantation d'un bloc EMIO routant un certain MIO vers le FPGA, etc...

Ces étapes seront effectuées essentiellement au travers des logiciels suivants :

*/répertoire\_d'installation/Xilinx/14.7/ISE\_DS/PlanAhead/bin/planAhead*

**Variables d'environnement** Pour pouvoir utiliser tranquillement la suite logicielle, faites dans le .bashrc :

source /répertoire\_d'installation/Xilinx/14.7/ISE\_DS/settings32.sh ou /settings64.sh

- PlanAhead va gérer l'ensemble de notre projet côté matériel, il tend à être remplacé par un autre outil promu par Xilinx nommé *Vivado* qui présente la même interface,
- XPS sera appelé par **planAhead** pour la spécification des bus et des ports.
- Enfin, le basculement vers le SDK et donc la partie software peut se faire directement dans planAhead.

## 3.2 Documentations

Vous trouverez tout au long du document, des notes vers les documents pertinents en fonction du chapitre. Néanmoins, les deux adresses générales qui peuvent vous permettre d'obtenir les schématiques de la carte, des tutoriels et des références.

<http://www.zedboard.org>

<http://forumzedboard.org>

## 4 Programmation du PS en l'absence d'OS (*Bare Metal*)

La première approche qui s'apparente la mieux à l'initiation à la programmation sur un micro-contrôleur "classique" consiste à développer une application n'exploitant que la partie processeur en l'absence de tout système d'exploitation. La compilation du programme C nécessaire à la définition de la séquence d'opérations exécutées se fait au moyen de l'outil propriétaire fourni par Xilinx.

### 4.1 Étape 1

Lancez planAhead au travers de l'exécutable qui se situe dans :

*répertoire\_d'installation/Xilinx/14.7/ISE\_DS/PlanAhead/bin/planAhead*

Nous allons spécifier une interface h/w la plus simple possible : CPU seul et aucune interaction avec le FPGA. Créez un nouveau projet dans votre répertoire de travail.



#### Create New Project

New Project Wizard will guide you through the process of selecting design sources and a target device for a new project.

Choisissez l'option **RTL Project**. **dire pourquoi, quel est sa particularité?** L'étape suivante nous permet de spécifier le langage HDL que l'on désire comme indiqué dans la figure 3 (a).

Les étapes suivantes nous permettent d'ajouter des fichiers sources (configurable IP, DSP composite, Embedded composite) ainsi que des contraintes (Add constraints) au projet lors de sa création. Sachant que ces fichiers et contraintes peuvent être ajoutés par la suite, nous choisissons de passer ces étapes en cliquant sur **Next** jusqu'au choix de la plate-forme sur laquelle nous allons travailler, **Zedboard Zynq eval** comme représenté en figure 3 (b). Après cette dernière étape un récapitulatif est proposé nous indiquant que les fichiers sources ne sont pas fournis. Cliquez sur **Finish** pour terminer la création du projet.

Le nouveau projet ainsi créé, PlanAhead ouvre son environnement de travail.

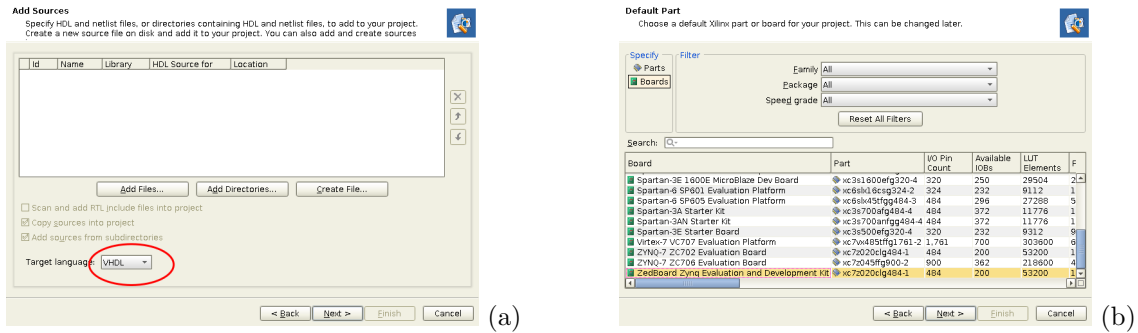


FIGURE 3 –

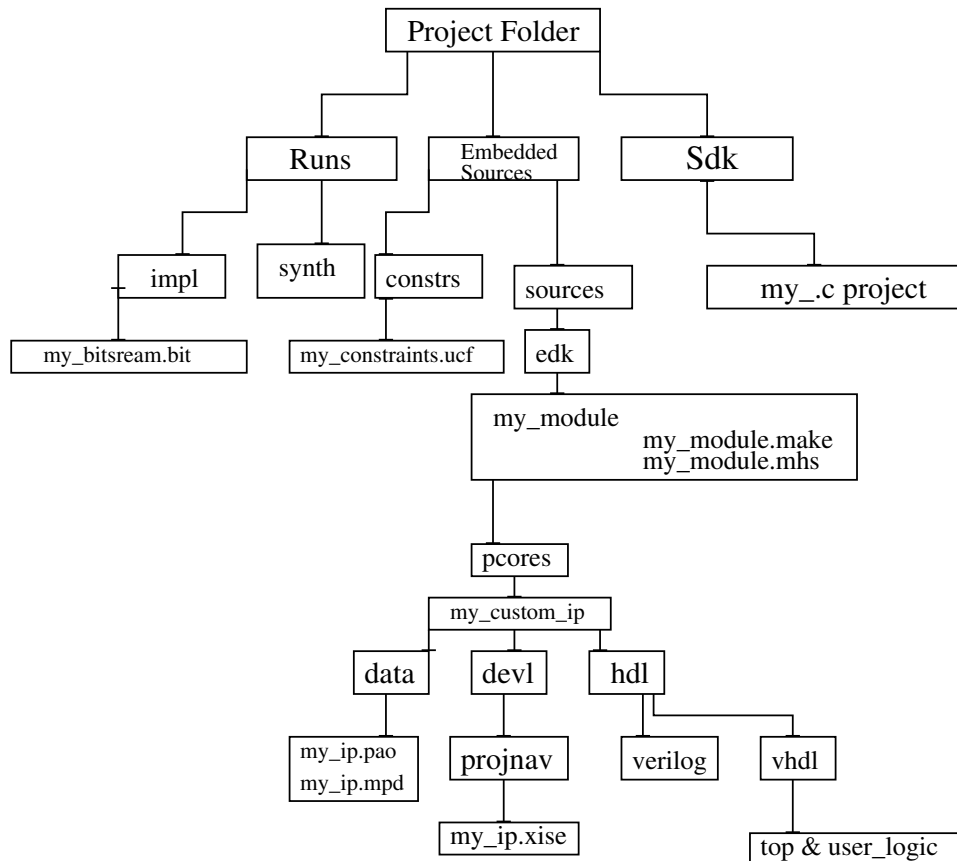


FIGURE 4 – Architecture typique d'un projet PlanAhead.

## 4.2 Étape 2

Afin de d'ajouter un processeur, dans le cadre Sources représenté en figure 5, cliquez sur le bouton



pour ajouter une source embarquée, autrement dit, un CPU. Cela peut également se faire par File/Add Sources.... Choisir Add or Create an Embedded Sources,

puis Create a Sub-Design... et lui donner un nom, dans notre cas proc1. Comme nous pouvons le voir dans la figure 6, le nom et la localisation de la source que nous venons de créer apparaît. Terminer la création de la source par Finish.

PlanAhead crée la source puis lance ce que nous appellerons XPS pour Xilinx Platform Studio. À cet instant, un premier message apparaît, indiquant que le projet est vide et nous demande si nous voulons

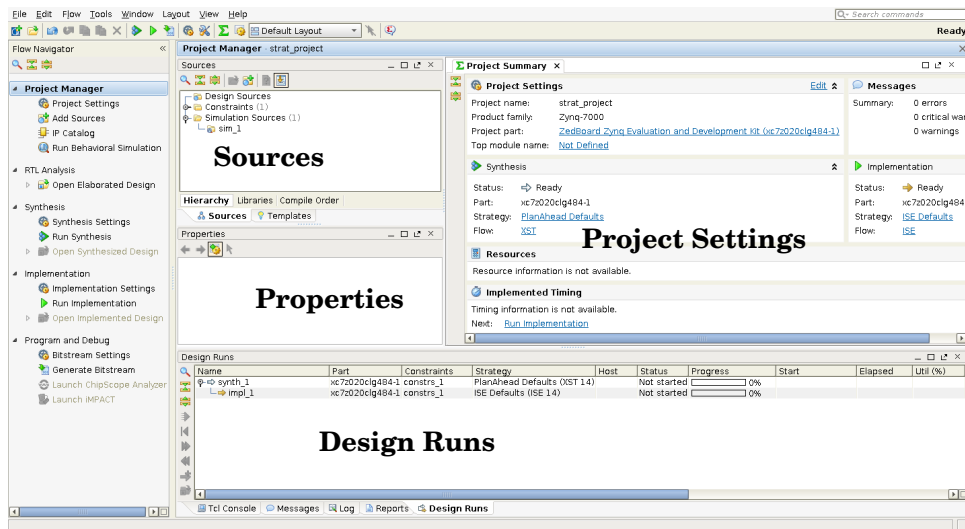


FIGURE 5 – Espace de travail de PlanAhead.

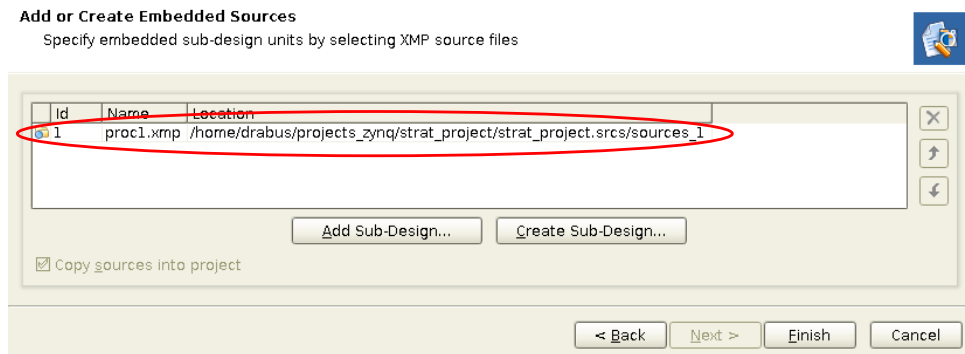


FIGURE 6 – Récapitulatif de la création d'un processeur.

créer un Base System using BSB Wizard, répondre oui. Ceci nous facilitera le travail pour la suite.

Différentes options sont proposées et nous laissons les choix par défaut. **Donner des informations sur les options de bases serait bien. ? (En s'appuyant sur les images, ou alors on vire les images mais il faut que le texte tienne la route) ?**

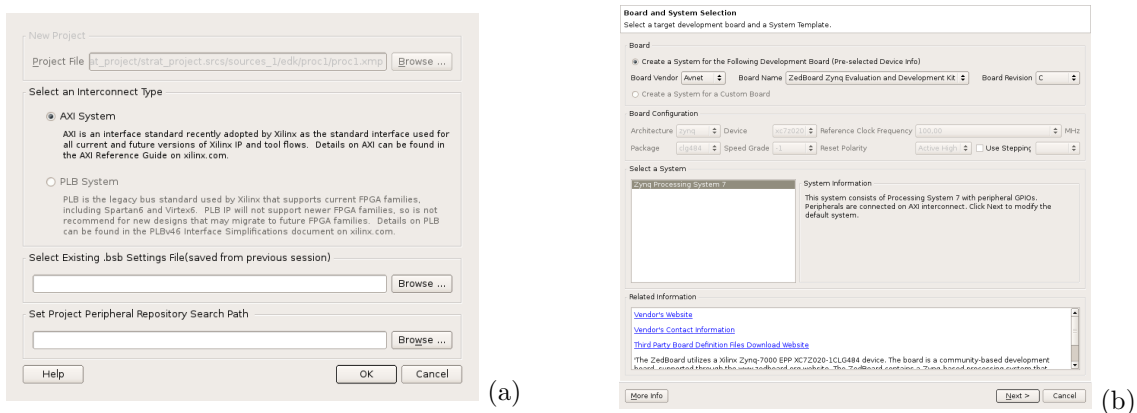


FIGURE 7 –

Une dernière fenêtre (**Peripheral Configuration**) s'ouvre afin de choisir les périphérique que nous voulons configurer. Nous faisons le choix de tous les supprimer de la liste. En effet, ces 3 périphériques que sont les Leds, switches et boutons, font partie des entrées/sorties de base pour le FPGA. Il est donc proposé de les intégrer directement, ce qui a pour effet de créer un fichier de contrainte comme nous le ferons plus tard, mais qui ne se trouvera pas dans notre arborescence `my_constraints.ucf`. Ce qui peut avoir pour conséquence de contraindre à nouveau ces pins et de déclencher un conflit. J'ai pris l'habitude de spécifier toutes les entrées/sorties systématique moi-même. D'autre part, nous n'en avons pas l'utilité pour notre première application. Cette étape là sera précisée en temps voulu : (section 4.5). Une fois la configuration du **Base System** terminée, l'environnement XPS est enfin accessible sous la forme représentée en figure 8.

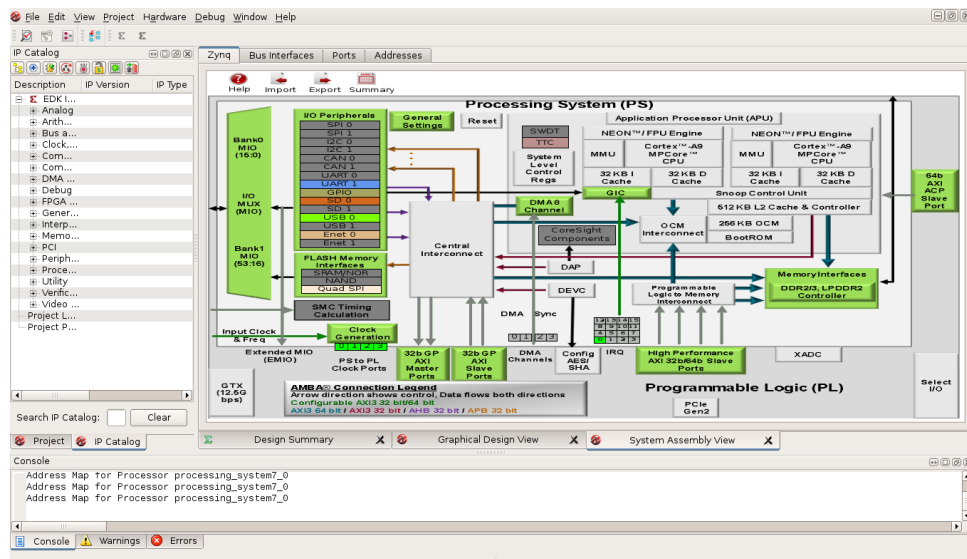


FIGURE 8 – Environnement XPS.

Cet environnement nous permettra de modifier les entrées/sorties du ou des processeurs, et surtout de configurer une architecture mêlant FPGA et CPU. À ce stade du tutoriel, nous n'avons donc pas besoin de cet environnement. Il sera présenté lors de la prochaine application.

### 4.3 Étape 3

Nous pouvons fermer l'interface XPS et revenir à PlanAhead. Dans le cadre **Sources**, nous remarquons que `proc1.xmp` est ajouté, qui est le module processeur que nous venons de créer. Faites clique droit sur ce module et choisissez **create top hdl**. Cela crée le fichier `nomdemodule_stub.bit` qui est notre bitstream : l'entité HDL de plus haut niveau de l'architecture. **Es tu sur de ce que tu dis ici ? il me semblait que l'on n'avait pas besoin du bitstream dans cette appli ? (que contient ce bit stream ?)**

— en fait je ne suis plus sur d'une chose : si l'export vers sdk est possible sans bitstream, il nous faudra tester ça en live lundi mais effectivement on ne flash pas le fpga dans le helloworld, pas besoin de .bit Cette manipulation donne lieu à une modification dans le cadre source comme nous le montrons en figure 9.

### 4.4 Étape 4

Ensuite, nous faisons appels à un nouvel outil qui se nomme SDK pour **Software Development Kit** qui va nous permettre de coder la partie software en C, la compiler puis de flasher nos codes ainsi que nos bitstream (fichiers de configurations du FPGA). Pour cela, allez dans **File/Export/Export Hardware for SDK**. Ne pas oublier ici de cocher la case **Launch SDK : Export vers SDK** va générer les fichiers d'initialisation dont nous disposerons dans notre projet C par la suite, ils sont basés sur notre

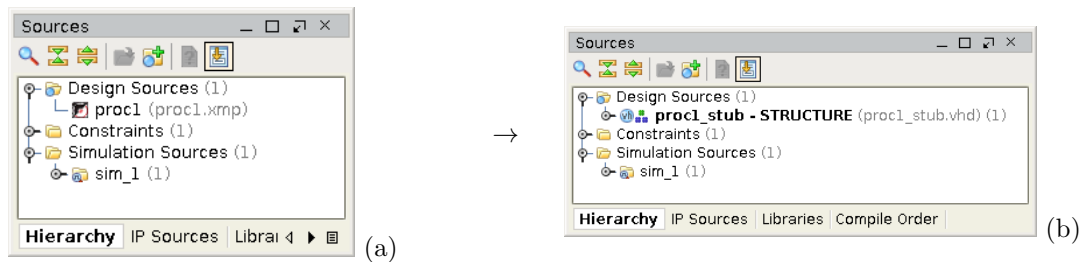


FIGURE 9 –


architecture hardware créée au travers d’XPS, et qui se retrouve dans le fichier module.mhs. Une fois l’environnement de travail ouvert, nous devons ajouter un template ”helloworld” à notre projet ce qui nous permettra de toujours disposer d’une base de travail simple et prête à l’emploi. En effet, comme vous pourrez vous en rendre compte rapidement, cette phase va inclure l’ensemble des fichiers sources mis à notre disposition par Xilinx pour notre partie Software. Vous pouvez les retrouver ici, et je vous invite à vous y référer par la suite lors de vos créations personnelles car chaque répertoire comprend un ensemble d’exemples qui seront très utiles :

mon\_repertoire\_d’installation/Xilinx/ma\_version/ISE\_DS/EDK/sw/XilinxProcessorIPLib/drivers  
 toujours se référer à la dernière version disponible.

Faites **File/New/Application Project**. Une fois fait vous êtes invité à donner un nom au template sans modifier les options, puis faites **Next**. Sélectionner **Hello Word** (le mettre en surbrillance) et faites **Finish**. L’environnement SDK réalise différentes compilations prenant en compte les différents fichiers sources. Notre projet est maintenant prêt à être compilé et flashé sur notre carte.

L’environnement SDK est présenté en figure 10. Nous remarquons différents cadres. Tout à gauche le **Project Manager** contient l’arborescence de notre projet, au centre un cadre servant à l’édition de fichier qui est associé au cadre tout à droite contenant les principales parties du fichier édité. Tout en bas une console dans laquelle SDK affiche différents messages. **Guillaume, est que la bibliothèque platform.h est ici de notre export vers SDK ou elle est faite avant ? plateforme.h est liée à notre export, elle initialise des différents composants liés au processeur ie : les cases cochées dans le zynq tab - peripherals dont on parlera plus tard**

Il est à noter que l’application que nous allons compiler envoie sur le port série les caractères **Hello Word**. Il faut donc connecter le port UART1 de la Zedboard sur le PC et ouvrir le port série avec la configuration suivante : bauds 115200 8N1. Afin de flasher la carte, nous allons utiliser le port JTAG. Pour cela nous devons mettre tous les jumpers parmi les 5 côtes à côte en position MIO. Une fois la carte alimentée, de part la position des jumpers, elle attend une séquence d’initialisation qui doit être chargée via le JTAG, ce sont les fichiers ps7init.c qui ont été générés **ou qui vont être générés par la**

**compilation ?**. Pour compiler et flasher le programme il suffit de cliquer sur l’icône lecture , mais il faut bien s’assurer que le template que nous avons créé est bien en surbrillance comme nous pouvons le voir dans la figure 10, où nous avons nommé notre template start.

**JTAG** Mise au point sur l’installation et l’utilisation des drivers pour pouvoir flasher correctement la carte :

déplacez vous dans l’arborescence d’installation vers :  
 repertoire\_d’installation/Xilinx/14.7/ISE\_DS/common/bin/linux/digilent  
 puis `sudo ./install_digilent.sh`

Pressez entrée pour toutes les étapes jusqu’à la dernière. Rajoutez dans le `.bashrc` :  
`export XIL_CSE_PLUGIN_DIR=/opt/`

Une fois le projet compilé et flashé sur la carte, le programme C est lancé et envoie sur le port série **Hello World**. Si le JTAG n’est pas reconnu automatiquement, vérifiez dans le menu **Xilinx Tools/Configure JTAG Setting** que **Auto Detect** est bien sélectionné. Une fois le projet flashé, les caractères **Hello World** sont envoyés sur le port série et sont donc visibles dans le logiciel que vous avez

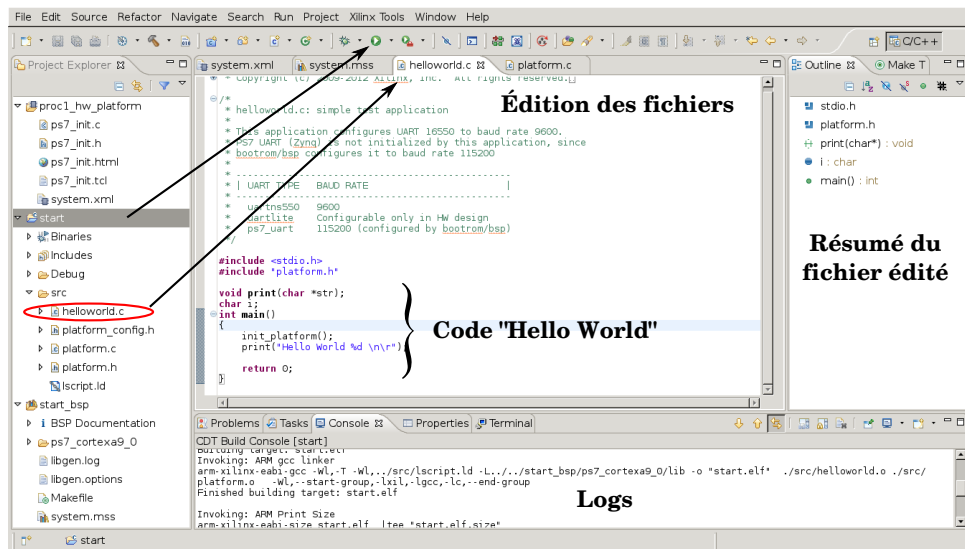


FIGURE 10 – Récapitulatif de la création d'un processeur.

utilisé pour ouvrir le port.

(David dit : la partie qui suit je ne sais pas où la mettre ) Une séquence d'initialisation est chargée dans le CPU au travers des différents fichiers `ps7.c` et notamment le fichier `ps7-init.c`. Ces fichiers appellent un certain nombre de séquence d'écriture dans des registres mémoires du cpu qui vont générer son initialisation (programmation des PLL par exemple). Cette étape dépend d'une spécification "hard" qui est ici la spécification la plus basique, à savoir : le CPU ne dispose que des entrées/sorties initiales. Une fois la séquence terminée nous disposons de l'ensemble des MIO avec lesquels nous pouvons directement interagir. C'est la configuration de base de la zedboard.

## 4.5 Application PS+PL

Je vais fusionner cette partie là avec la partie que je viens d'écrire juste en dessous : on garde nos explications pour chacune d'elle, mais l'application créée est "axi\_gpio". Plus simple et plus claire pour commencer.

Dans ce nouvel exemple nous allons créer une application permettant d'agir sur des périphériques (LEDs et boutons) via un programme C flashé dans le CPU. Cela nécessitera la génération d'un bitstream qui sera l'interface entre notre PS et PL.

Respectez scrupuleusement les deux premières étapes décrites dans la première application. Dans un premier temps nous allons créer un nouveau projet comme précédemment (étape 1) en ouvrant **PlanAhead** et en faisant **Créer un nouveau projet**. Ensuite ajouter un module processeur (étape 2) en cliquant sur **Add sources** et en sélectionnant **Add or Create an Embedded Sources**.

Contrairement à la première application qui n'utilisait pas de périphérique à travers le FPGA, ici nous devons indiquer quels sont les LEDs et bouton que nous voulons utiliser. Cela se fait dans l'environnement XPS représenté en figure 8. Dans cette fenêtre nous allons définir et programmer notre environnement pour le CPU et sa relation avec le FPGA. À gauche de la fenêtre se trouve la bibliothèque d'IP : **Intellectual Properties**. Ce sont des blocs fournis par Xilinx et que nous pouvons implémenter dans notre architecture afin d'utiliser les périphériques souhaités. Nous pouvons également réaliser nous même un IP qui l'on nommera **User IP**. Au centre, nous avons trois principaux onglets qui se situent en dessous :

- Design Summary,
- System Assembly View,
- Graphical design view

Comme son nom l'indique, **Design Summary** résume le module que nous sommes en train de créer.



L'onglet **Graphical design view** permet de visualiser la connexion entre les différents blocs et bus que nous créons. L'onglet principal **System Assembly View** comporte quatre sous onglets qui se situent au-dessus. Le premier sous-onglet **Zynq** nous donne la représentation en schéma bloc de notre architecture. Chaque bloc vert peut être modifié lorsque l'on clique dessus. **il y a t il une doc qui décrit les différence entre chaque couleur ? pas à ma connaissance** Le sous-onglet **Bus interfaces** nous permet de connecter sur le bus AXI nos différents composants. En général, nous utiliserons des connexions générées automatiquement, ce qui aura pour effet de placer et de lier les différents sous-composants, les architectures peuvent devenir rapidement complexes. Si il y a un besoin spécifique pour créer sa propre architecture manuellement, alors il faudra tout lier soi-même dans cet onglet.

Le sous-onglet **Port** va regrouper l'ensemble des entrées sorties systèmes. Dans notre application, nous allons implémenter un IP AXI GPIO qui nous permet de contrôler un nombre paramétrable d'entrées/sorties au travers du Bus et donc vers/depuis le FPGA. Si on imagine 8 Leds en sortie, alors dans l'onglet **Ports** nous aurons `user_ip.Leds_top_0 UNCONNECTED`. Il nous suffit alors de faire clic droit - Make External pour que le système en face un port Externe. Toutes nos étapes dans XPS résultent en notre module.mhs. Ainsi, le make external va créer un PORT dans notre entité Custom\_IP.Leds\_top dans ce fichier la, et va le lier à une variable leds\_pins. C'est cette variable qu'il nous faudra contraindre avec un fichier .ucf, pour cela il faudra faire "ajouter une contrainte" dans PlanAhead, au lieu de faire ajouter une "embedded source".

Enfin, le sous-onglet **Bus Interface** nous permet de voir - modifier - générer l'ensemble des adresses mises en jeux dans notre application. Notons le petit bouton situé juste en haut à droite qui permet de générer les adresses de manière automatique.

Dans le catalogue d'IP, *Intellectual Property*, choisissez un bloc AXI-GPIO et ajoutez le. Dans ses paramètres, choisissez un nombre de bits comme étant de 8 : ce seront nos 8Leds. A noter que si l'on souhaite disposer d'entrées et non de sortie, il nous faudra spécifier "Input Only". Comme toujours, on pourra par exemple spécifier une adresse spécifique pour cet IP. Validez et choisissez la connexion automatique, ce qui a pour effet d'interfacer notre bloc convenablement sur le bus. On retrouve alors dans nos onglets **Ports** et **Tabs** sous le nom AXI-GPIO-0, dans la représentation graphique vous devez voir un bloc représentant le processeur, un bloc AXI Interconnect qui a été généré automatiquement et qui va s'interfacer entre l'AXI.GPIO, et le bloc AXI.GPIO.

Fermez xps et nous revenons sur planAhead.

cliquez droit sur votre `module.xmp` et générez le `top.vhdl` lui correspondant.

Ouvrez un terminal et parcourez votre projet vers le répertoire du module.mhs comme explicité dans nos premières pages.

Ici vous trouverez les fichiers les plus importants générés pour la partie h/w. le fichier `nomdumodule.mhs` est celui de la spécification h/w. Si vous l'ouvrez vous pourrez lire l'ensemble des ports de votre CPU, les paramètres spécifiés lors de l'étape sous XPS. Vous avez créé une interface AXI.GPIO elle est instanciée par un `BEGIN axi_gpio`. On peut y lire notamment les adressages sur le BUS et nos entrées/sorties système, les clocks de références etc..

Noter la variable terminant par `_pins` se référant à votre AXI.Gpio.

Dans planAhead faites **Add Sources** et choisissez **Create Constraints** en nommant par exemple le fichier `proc.ucf`. Ce fichier se trouve alors dans l'arborescence à l'emplacement `my_module.ucf`. Modifiez ce fichier avec l'éditeur de texte pour contraindre nos 8bits du bloc AXI Gpio à correspondre physiquement à nos 8Leds. Pour ce faire la nomenclature typique est `NET axi_gpio.0_top_pins[0] IOSTANDARD=LVC MOS18 — LOC = T22`; pour la led0 par exemple. `j[i]` se réfère au bit i du vecteur `Leds[7 :0]` Les Leds sont situées en : T22 ; T21 ; U22 ; U21 ; V22 ; W22 ; U19 et U14.

Resortez du fichier et retourner dans l'arborescence, aux sources hardware de notre module processeur.

Le répertoire contient un fichier `nomdemodule.make` qui est appelé lors de toutes les générations de fichier du logiciel planAhead.

Pour que nos compilations se passent bien, il nous faut créer un lien symbolique de `gmake` vers `make` :

```
sudo ln /usr/bin/gmake /usr/bin/make
```

Faites `gmake -f monmodule.make netlist`

La netlist va être générée.

Pour des raisons encore indéterminées, il se peut que l'étape XPS ait généré de mauvaises valeurs pour

les CLOCKS de référence. Si la netlist n'est pas générée et vous donnez des erreurs avec des fréquences d'horloges, c'est alors le cas. Retournez dans le `module.mhs` et spécifiez pour les 4 horloges de références les valeurs suivantes :

CLK0 = 100000000 (100MHz clk ref),

CLK1 = 142857132 (allez savoir..),

CLK2 = 50000000 (50MHz),

CLK3 = IDEM

Enregistrez.

Faites un `gmake -f monmodule.make netlistclean`

puis enfin `gmake -f monmodule.make netlist`.

Revenez dans planAhead et lancez la génération du bitstream qui normalement doit se passer sans encombre. Cliquez sur **fichier** puis **export to sdk** en choisissant l'option export h/w et cochez launch sdk.

Vous trouverez le code C pour une interaction avec l'axi gpio en page...

Vous pouvez vous référer dans le repertoire des drivers Xilinx cité en debut de chapitre, pour avoir des exemples d'utilisation des XGpio. Pour vous repérer avec la nomenclature Xilinx : X[entitee]\_Ps : driver software pour le processeur X[entitee] : driver software pour un IP Xilinx s'interfaçant sur le bus au fpga.

Avant de flasher le code C nous devons flasher le fpga avec le bitstream.bit que nous avons généré lors de l'étape précédente. Pour cela, cliquez sur Xilinx Tools program FPGA et pointez dans l'arborescence vers le fichier .bit. Une LED bleue est allumée indiquant qu'un bitsream valide a ete chargé. Ouvrez le terminal avec l'UART1 comme précédemment : 115200baud 8N1.

Flashez le code C comme lors de la première application. Nous tappons à un ensemble d'adresse qui nous permet d'avoir le contrôle des sorties du FPGA au travers du bloc de Xilinx pour GPIO.

## 4.6 Application PS+PL axi\_gpio

Cette application remplacera a terme l'appli précédente.

**Apperçu** Nous allons créer l'application suivante :

Nous implantons un ip proposé par Xilinx dont le rôle est de réceptionner les signaux du bus AXI selon la norme SLAVE, de permettre le contrôle depuis le processeur de jusqu'à 32 entrées/sorties.

**Partie h/w** Ouvrez planAhead et créez un nouveau projet. Ajoutez une source embarquée nommée proc par exemple. XPS s'ouvre. Dans la bibliothèque d'ip, inclure un "General Purpose IO" axi\_gpio. Cliquez oui. Dans les options choisissez un nombre de bits de 8 : nous allons contrôler 8Leds. Notez que si l'on avait voulu lires les boutons par exemple, il nous aurait fallu cocher la case "All Inputs". Validez et choisissez une connexion automatique avec processing\_system\_7 qui est l'architecture du processeur que nous somme entrain de spécifier.

Dans le sous-onglet graphical view vous pouvez voir alors deux blocs qui viennent d'être ajoutés :

axi\_Interconnect

axi\_gpio

Axi Interconnect a été ajouté automatiquement : on ne peut avoir une connexion axi sans avoir de bloc interconnect. La connexion automatique a coché l'option "General Master Port" dans le general settings du Processeur situé :

Puis a ajouté un axi interconnect, disponible dans l'ip catalog sous : "bus and bridge".

Dans l'onglet Bus interface, vous pouvez voir que la connxion sur le bus "S\_AXI", pour slave, de notre ip axi\_gpio a été reliée à l'interconnect. Notre processeur possède également un Master GP, pour General Purpose Master Port.

Il est également possible de faire les connexion à la mains : il faut ajouter un interconnect, cocher l'option GP\_0, les relier dans le bus interface, et faire également les étapes suivantes dans les onglets PORTS et ADDRESSES :

dans l'onglet port vous retrouvez (en connexion automatique) : Sous axi\_gpio nous disposons d'un port Gpio\_O qui est en fait un vecteur de 8bits maintenant, et le système, grace à la connexion automatique

## ZYNQ

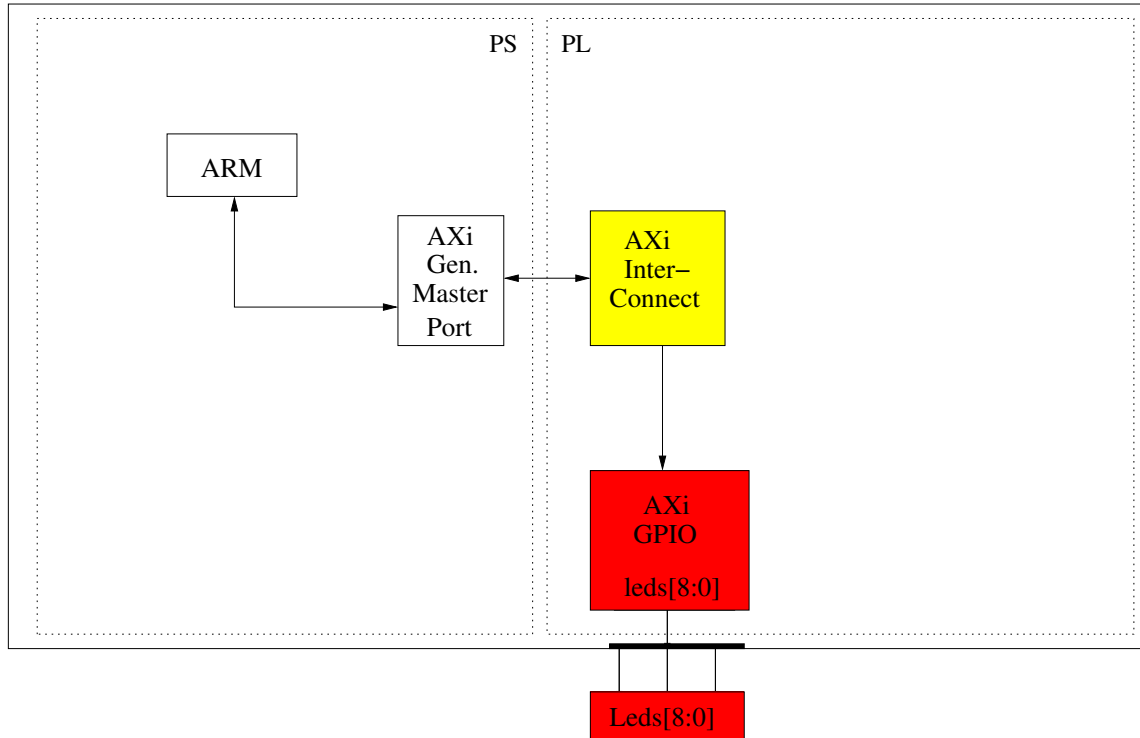


FIGURE 11 – axigpio

toujours, la paramétrer comme entrée/sortie du système :

on fera cela, par la suite, avec un clic droit : "Make External".

Ce qu'il fait qu'il se retrouve, toujours dans l'onglets PORTS, sous processing\_system\_0 déclaré en variable axi\_gpio\_0\_pin [7 :0].

C'est cette variable qu'il va nous falloir contraindre dans un fichier de contrainte, aux broches physiques du FPGA, correspondants aux LEDs de la carte. Enfin dans l'onglets ADDRESSES, vous pouvez voir que notre bloc axi\_gpio s'est vu affectée une adresse, qui est comprise dans l'intervalle de notre interconnect. Autrement dit, une connexion manuelle devra vérifier l'ensemble de ces points pour être valide. Quand nous feront des "custom ip" - ou des ip "persos", nous devront notamment spécifier, via "Make External", nos entrées/sorties physiques, et les contraindre dans un fichier .ucf.

Vous pouvez fermer XPS et vous retrouver à nouveau dans planAhead.

Cliquez à nouveau **add sources** et choisissez cette fois "add or create constraints".

L'arborescence, my\_constraint.ucf vient d'être créée. Nous allons modifier ce fichier pour contraindre notre PORT à correspondre aux broches physiques des leds. Pour se faire : récupérer le nom de la variable exacte, dans le fichier proc.mhs.

On cherche en l'occurrence, le PORT axi\_gpio ...\_pin

Dans notre application, il s'agit d'un vecteur de 8bits, il nous faut tous les contraindre pour éviter des erreurs de compilation, pour se faire, dans le proc.ucf :

```
NET axi_gpio_0_pin[0:7] IOSTANDARD=LVC MOS18 — LOC = T22;
```

contraint le bit 0 de notre port à la led 0 qui correspond à la pin T22.

Faire ceci pour l'ensemble des bits : T21, U22, U21.. jusqu'à U14.

Dans notre répertoire source, faites un **make -f proc.make netlist** , pour générer la netlist. Une fois que c'est fait, générer le bitstream via planAhead.

**Conclusions sur l'h/w** Nous pourrions travailler en choisissant d'implanter des connexions automatique, et ce, même sur des architectures beaucoup plus complexes, ce qui nous évitera de nous soucier des interconnexions : tant que nos IP respectent une certaines normes AXi, on pourra toujours les connecter à un interconnect. Nous verrons en temps et en heure comme le respect des protocoles se fait.

Enfin la partie contrainte est fondamentale pour la réussite de la compilation de nos projets.

**Partie s/w** Notre bitstream est généré et se retrouve dans l'arborescence, dans le sous répertoire qui vient d'être crée `Runs` puis `impl_1` sous le nom `proc_stub.bit`. C'est ce fichier que nous devons flasher sur la carte, avant de flasher n'importe quelle application software.

Dans `planAhead`, faites fichier, export vers SDK, cochez export hardware (à faire une seule fois) et launch SDK, qui va nous lancer l'interface graphique. Notez que l'on peut faire également `make -f proc.make exporttosdk`.

On se retrouve alors dans `eclipse`, qui va nous permettre de flasher le bitstream et le fichier `.elf`. Faites un nouveau projet dans File new application project, nommez le "gpio" et cliquez OK pour un modèle helloworld. Pour ce qui est du baremetal, il est plus aisé d'utiliser le modèle helloworld, dans le sens où le sous répertoire ainsi créée : `gpio_bsp`, inclue l'ensemble des librairies de Xilinx qui nous permettront de coder notre application. Pour les retrouver : cité dans l'application précédente

Flasher le bitstream.  
Flasher le code source.

**Programme C** Le programme utilise la structure `XGpio`, vous pouvez vous inspirer des exemples du répertoire Xilinx, le `#include "xparameters.h"` nous permet de retrouver notre constante `AXI_GPIO_0` définie depuis l'export h/w vers SDK.

## 4.7 Les trois interactions GPIO possibles

À partir de là nous sommes capables d'écrire une application PS+PL en interfacant les GPIO de toutes les manières que nous offre la carte :

MIO - EMIO - AXI\_GPIO. Voici une application que j'ai écrite avec les caractéristiques suivantes : On a un MIO BTN8 qui sera une des entrées système. On a déjà vu que les MIO sont implantés automatiquement tant qu'ils sont cochés dans le IO Peripherals Tabs de XPS.

On a un block AXI\_GPIO de 8bits pour les LEDs qui seront une des sorties du système.

On a un block AXI\_GPIO d'1bit pour l'entrée d'un des boutons.

Ouvrez un nouveau projet et commencez une spécification de la partie h/w.  
Créez une source embarquée.

Dans l'IP Catalog à gauche, spécifiez une interface axi gpio sur le channel1 de 8bits. Choisissez un interfacement automatique. Spécifiez une interface axi gpio sur le channel1 de 1bit en cochant "ALL INPUT", interface automatique également. Ne décochez pas les MIO dans le IO Peripherals Tabs (assembly view).

Dans l'onglets **Ports**, dans les blocs `axi_gpio` faites un clic droit Make External sur les `axi_gpio_pins`  
Fermez XPS.

Créez une nouvelle contrainte pour ces nouvelles entrées/sorties vers les pins pertinants : `add constraints .ucf`.

Pour retrouver les noms de variables à contraindre, regardez dans le fichier `module.mhs` dans les **PORTS**. Nous cherchons les noms en `_pins` Le fichier contrainte utilisateur est unique et se trouvera toujours dans (une fois créée..) : `/repertoireduprojet/nomdumodule.srcs/constrs_1/new/monfichier.ucf`

```
NET axi-gpio-0-pin<0> IOSTANDARD=LVCNMOS18 | LOC = xx
NET pins des LEDS: <!!> logic cmos 33
NET pins des 5BTNS: <!!> logic cmos 25
```

NET pins des Sw: <!=> logic cmos 25 \newline

Les bits des vecteurs sont appelés avec les [i]. LOC est la localisation sur le FPGA, d'après le document \*\*. Juste derrière NET vous devez inscrire le nom exact qui est celui du top.vhd. Le top vhd est : /repertoireuprojet/nomdumodule.srscs/sources.1/edk/nomdumodule/nomdumodule\_stub.vhd Vous pouvez aussi retrouver ces noms la au tout debut du fichier nomdumodule.mhs dans les specifications PORTS, ils portent les noms que vous leur spécifiez dans XPS avec l'extension \_pins.

generez le top et le bitstream.

Exportez to SDK. Importez un template C Helloworld pour avoir les bons #includes.

Les axi\_gpio sont déclarés comme XGpio. Le MIO est déclaré comme XGpioPs. Le code donné en annexe vous donne une application où chacune des leds clignotent l'une après l'autre. Tant que le bouton est pressé alors les LEDS clignotent dans le sens opposés Le sens de clignotement est dicté par le type du dernier bouton pressé ( AXI ou MIO ?).

## 4.8 Application bare Metal et Custom IP

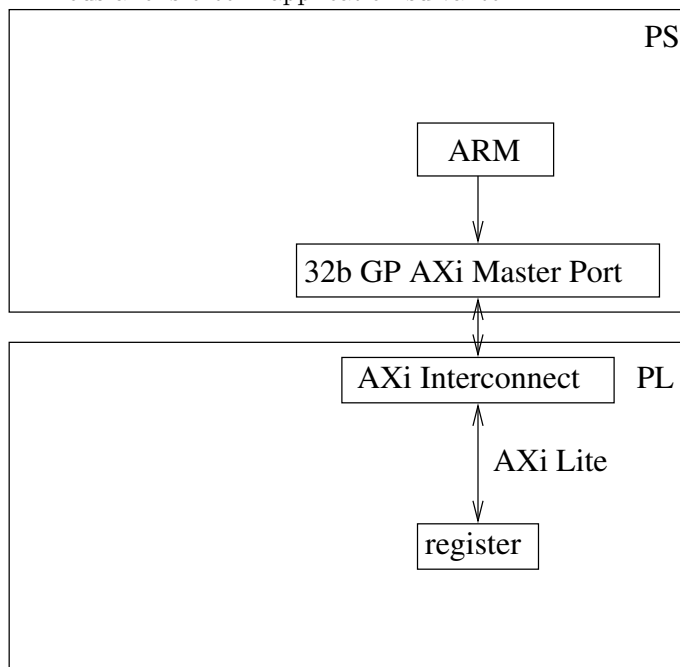
Nous allons créer une application implantant un module vhd dans la partie PL et en le pluggant sur le bus AXi.

Pour faire cela, on disposera d'une interface de programmation dans planAhead, dans laquelle on spécifiera la norme AXi que l'on souhaite utiliser.

Xilinx nous propose de nous aider en définissant un certains nombres de registres mémoires accessibles en écriture et ou en lecture depuis le CPU vers le FPGA. On pourra s'en passer par la suite, mais cela permet de cacher la compréhension de la communication mise en place sur le bus.

## 4.9 Un simple registre memoire

Nous allons créer l'application suivante :



L'objectif étant que le CPU vienne écrire puis lire la valeur contenue dans le registre. Créez une interface CPU avec planAhead, n'oubliez pas d'enlever les 5BTNS, 8SWs et 8LEDS de la zedboard. Une fois cela spécifié, cliquez dans les menus en haut *add custom peripheral* et suivez le guide. Nous cherchons à créer l'interface la plus simple possible, cela comporte notamment les étapes suivantes :

Specification du protocole AXi souhaité :

AXi4Lite, si vous vous êtes déjà référés à la documentation mentionnée pour le protocole AXI, vous savez qu'il s'agit du mode de communication le plus basique. On dispose d'une possibilité d'écriture et de lecture depuis le CPU, les données et les adresses sont sur 32bits.

Le nombre de registres :

Ici nous spécifions le nombre de case de 32bits dans lesquelles nous allons pouvoir venir lire et écrire depuis la partie software.

AXiSlave - Master : spécifier le comportement du bus *côté FPGA*. Slave : le FPGA ne fait *que recevoir* les adresses des actions d'écritures et de lecture. Master : possibilité de passer une adresse au CPU depuis le FPGA.

Interaction Software :

Ici nous choisissons quelles actions vont être possibles depuis notre CPU selon les templates Xilinx qui seront alors générés. Write - Read donne l'accès en écriture et en lecture au cpu. Reset : le CPU disposera d'une certaine adresse qui va redémarrer le bloc HDL directement en invoquant la fonction exportée dans le SDK. Encore une fois tout nous est caché par la couche de template Xilinx que ce soit au niveau HDL et Soft.

Choisissez donc Write Read et Reset et décocher phase line.

Cochez Generation du template, qui va générer une petite interface C que l'on pourra alors importer dans notre SDK pour venir directement taper aux adresses des registres spécifiés.

Si vous cochez la case du dessus (aide a la synthese XST) vous disposerez alors d'un projet ISE (.xise) qui implémentera alors les deux blocs HDL que nous allons invoquer tout de suite après, et qui vous permettra de réaliser des simulations sur le Bus, de coder votre IPCore et de déboguer. Ce projet se trouve alors dans le répertoire projnav comme spécifié au début de notre tutoriel.

Tout ce qui touche à la generation des IPs se trouve dans notre répertoire Pcores (cf arborescence projet). `monprojet/monprojet.srscs/sources-1/edk/monmodule/pcores/jnomdel'ip`

Le projet eventuellement généré se trouve en `/jnomdel'ip/devl/projnav`

Les fichiers HDL se trouve dans le repertoire HDL. Le répertoire Data regroupe deux fichiers `.mpd` qui va spécifier toutes les entrées et sorties de notre IP, ainsi que la norme AXi suivie, `.pao` qui va récapituler l'ensemble de nos blocs hdl invoqués, une sortes de `#includes`.

Le driver software généré pour notre IP se trouve dans `monprojet/monprojet.srscs/sources-1/edk/monmodule/pcores/jnomdel'ip`

Dans ce répertoire vous trouverez un fichier de "driver" pour notre cpu qui est le "monipcore".h.

Une fois le process "add custom peripheral" terminé, vous disposez d'un nouvel IP ajoutable depuis l'IP Catalog. Si vous laissez l'application tel quel, vous pouvez déjà utiliser vos registres mémoires depuis le CPU en écriture et en lecture.

Pour comprendre un peu ce qui se passe, le logiciel a généré un repertoire au nom de l'ip qui contient : un `top.vhd` et un `user-logic.vhd`.

Le top est celui dans lequel on ne codera jamais. Il instancie des templates Xilinx qui décodent la norme Axi spécifiée. On n'y fera que répercuter nos différentes entrées/sorties que nous rajouterons. Il faudra alors les relier correctement dans l'onglet PORTS.

Le user-logic propose une interface où il nous sera possible d'instancier notre bloc, de disposer des signaux du bus, et de développer une petite application spécifique.

Ajoutez donc un exemplaire de votre IP et choisissez une connexion automatique. Fermez XPS et générez la netlist comme précédemment.

Générez le bitstream et faites un export vers SDK.

Faites un `cp monipcore.h /home/usr/monprojet/monprojet.sdk/sdk/sdk-export/src` . Le driver se trouvant a l'endroit spécifié précédemment.

Rafraichissez le sdk comme dans eclipse et il devrait apparaitre s'il est dans le bon repertoire maintenant.

Etudiez un peu le fichier en question, nous pouvons voir que nous disposons d'une fonction d'écriture et de lecture. Si vous avez une erreur d'include avec le fichier `xbasic-types.h` cité comme introuvable :

Allez dans votre repertoire ISE d'installation et faite une recherche pour ce fichier. Relevez son chemin exact et modifiez dans le SDK le `#include "xbasic-types.h"` par `#include "lerepertoireexact.h"`

normalement, la compilation est un succes.

Nous sommes donc capable d'écrire et de lire une valeur depuis le CPU dans un registre qui est un module vhd qui nous avons implanté dans notre partie PL.

#### 4.10 Aller plus loin

notre driver.h defini plusieurs choses importantes. Tout d'abord deux fonctions lecture et écriture. Si vous vous souvenez lors de notre interfacage d e bus nous avons pu noter la plage d'adresse mémoire qui a ete affectée à notre IP. Par exemple 0x4640 0000 (Base-Address) -0x4640 FFFF ; Notre registre ne dispose que d'une seule case de 32bits, aussi nous ne pouvons écrire/lire qu'un seul emplacement. Si nous en avons plusieurs nous procéderions comme ceci :  
Registre0 = Base-Address. Registre1 = Base-Address+offset. RegistreN = Base-Adress+N\*offset.

### 5 Un transcodeur binaire

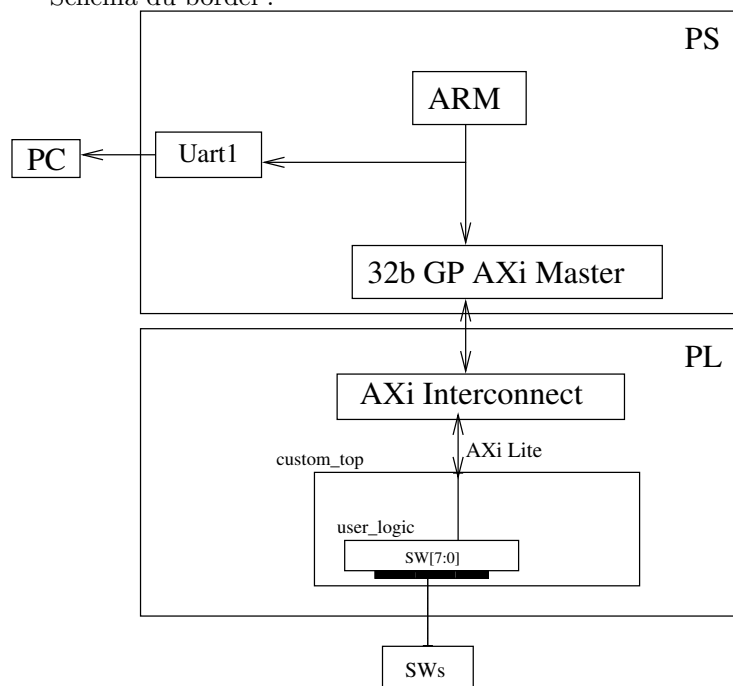
Je vous propose de recréer une petite application que j'ai ecrite enfin de decouvrir un peu l'interface proposee dans le user-logic.

Cette interface est lourde et pénible, on arrivera rapidement à se passer des fichiers modèles invoquées par Xilinx, si ce n'est en les supprimant, en les bypassant complètement et en codant juste à côté.

En attendant, l'interface planAhead nous permet d'avoir une communication de 32 registres de 32bits accessibles en écritures et en lecture depuis le CPU relativement facilement.

Nous allons créer une interface vhd un peu plus complexe. Cette fois-ci des entrées utilisateurs seront ajoutées directement a notre IP : SW[7 :0]. Un mot binaire sera écrit dans le registre à partir de ces switchs. Le Cpu viendra lire cette valeur lorsqu'un bouton sera pressé (MIO).

Schema du bordel :



Reprenez le tutoriel précédent et générez un custom peripheral avec un registre mémoire de 32bits selon la norme AXi Lite.

Une fois qu'il est disponible dans votre IP Catalog, ouvrez les fichiers user-logic et top dans un éditeur de texte. On peut également choisir de travailler en ouvrant le projet ise spécifié précédemment.

Nous allons rajouter un bout de code dans le fichier user-logic.vhd, d'ailleurs, étant donné que je suis fort sympathique, le voici en annexe X.

On ajoute un vecteur d'entrée de 8bits qui seront les switches de la Zed.

La valeur du registre mémoire prend la valeur d'un signal intermédiaire. Ce signal intermédiaire vaut à chaque instant, la valeur des switches.

Ainsi en lisant depuis le CPU, on viendra lire la valeur binaire écrite depuis les boutons poussoirs.

Ajouter enfin les nouvelles entrees au fichier TOP au dessus du user-logic, et mappez convenablement les entrees SW.

Notre application est simple et seuls les fichiers top.vhd et user\_logic.vhd entrent en jeux *Pas de modification* du fichier .pao, puisqu'ils y sont déjà. Si à l'avenir, il nous faut ajouter des blocs HDL, il faudra les spécifier dedans, car le .pao est important dans l'étape suivante.

Dans XPS, faites un "add custom peripheral" et choisissez import existing peripheral. Entrez le nom précédemment choisi et cliquez oui sur le fait de vouloir l'écraser. Un nouveau répertoire est créé, nommé data\_old, il regroupera votre ancien .mpd et .pao.

Dans la rubrique suivante, choisissez l'option .pao et pointez vers le pao dans le répertoire data.

NB : ON PEut choisir de travailler aussi diféremment : cochez l'option - à tester.

## 5.1 Pour aller plus loin

Arrêtons nous un instant et étudions ces deux blocs avec lesquels nous avons intéragi. Nous avons vu que le fait de générer un "custom IP" nous a donné deux fichiers, un générique portant le nom notre\_IP\_top.vhd et un user\_logic.vhd destiné à être modifié par une logique écrite par l'utilisateur.

Le fichier top instancie des templates Xilinx qui decode la norme axi lite spécifiée. Ces blocs nous redistribue un ensemble de signaux, qui sont mis à notre dispositions pour nos développement futurs.

## 6 FPGA Avancé et AXi Protocol

Dans cette partie nous allons commencer à utiliser le Zynq d'une manière plus avancée, il nous faudra comprendre les différents protocoles de communications mis en jeux.

On abordera la mise en place d'interruption du FPGA vers le CPU et vice-versa.

Les transferts de type streaming à débits plus larges, dans les deux sens.

## 7 Protocole AXi : AXi Lite Slave

### 7.1 Étude préliminaire

Pour étudier la norme AXi Lite veuillez vous référer au graphique suivant :

Nous nous intéressons aux bits et vecteurs dont nous disposons aux seins de nos IP : ce sont situés en bas sur le graphique

Bus2IP et IP2Bus où les premiers sont ceux que nous recevront, et les derniers sont ceux que nous devons envoyer.

On peut voir que le reset est actif a l'etat bas, pour la suite j'ai pris l'habitude de le remettre en logique habituelle au sein du fichier user\_logic.vhd.

Les bits CS pour Chip Select et RNW pour Read N Write sont les plus importants pour déterminer quelle action est en cours.

RNW est le plus simple : un état *HAUT* pour une requette de .

un état *BAS* pour une requette de .

Quand je parle de requette, ce sont les requettes émises par l'un des processeurs de la zedboard.



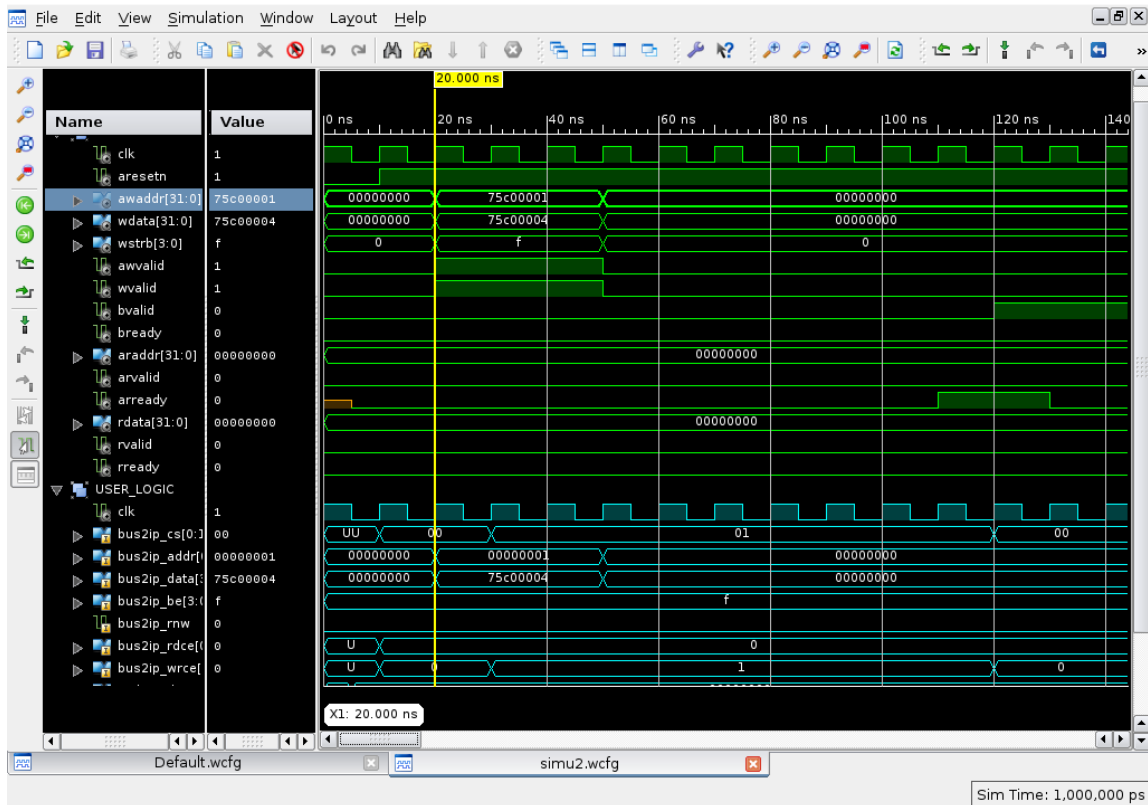


FIGURE 12 – AXi Lite Single Write Event

Le CS est en fait un nombre de bits qui dépend du nombre de registres choisis lors de la création de l'IP et du choix de l'action reset possible ou non depuis le processeur. Pour avoir un nombre de bits constant pour le CS et avoir un code générique pour la suite, j'ai choisi de toujours spécifier, lors de la création de l'IP dans XPS :

Reset possible et UN SEUL registre mémoire accessible depuis le CPU.

Ainsi nous aurons 2 bits pour le CS et lorsque le bits de poids fort CS(1) sera à l'état HAUT alors le CPU s'adressera bien à notre IP.

Nous disposons de l'adresse qui est sur 32bits : `Bus2IP_Addr` qui nous permettra de fixer des registres et pouvoir coder des actions différentes dans le FPGA.

Enfin, pour ne pas bloquer le bus, nous devons faire deux *Ack* :

Acknowledge de lecture en mettant `IP2Bus_Ack` à l'état 1

Acknowledge d'écriture en mettant `IP2Bus_Ack` à l'état 1

`IP2Bus_Error` sera toujours maintenu à l'état bas.

Conclusion : Les bits CS et RNW nous permettent de déterminer si le CPU s'adresse bien à notre IP et si la requête est une lecture ou une écriture.

NB : Les vecteurs CS et Addr sont en *LSB First* et le bit Reset est actif à l'état Bas.

## 7.2 Code HDL pour AXi Slave

Prenons l'exemple de mon driver `zynqled` qui permet de piloter les leds du fpga depuis linux.

## AXi Lite Single Write Event

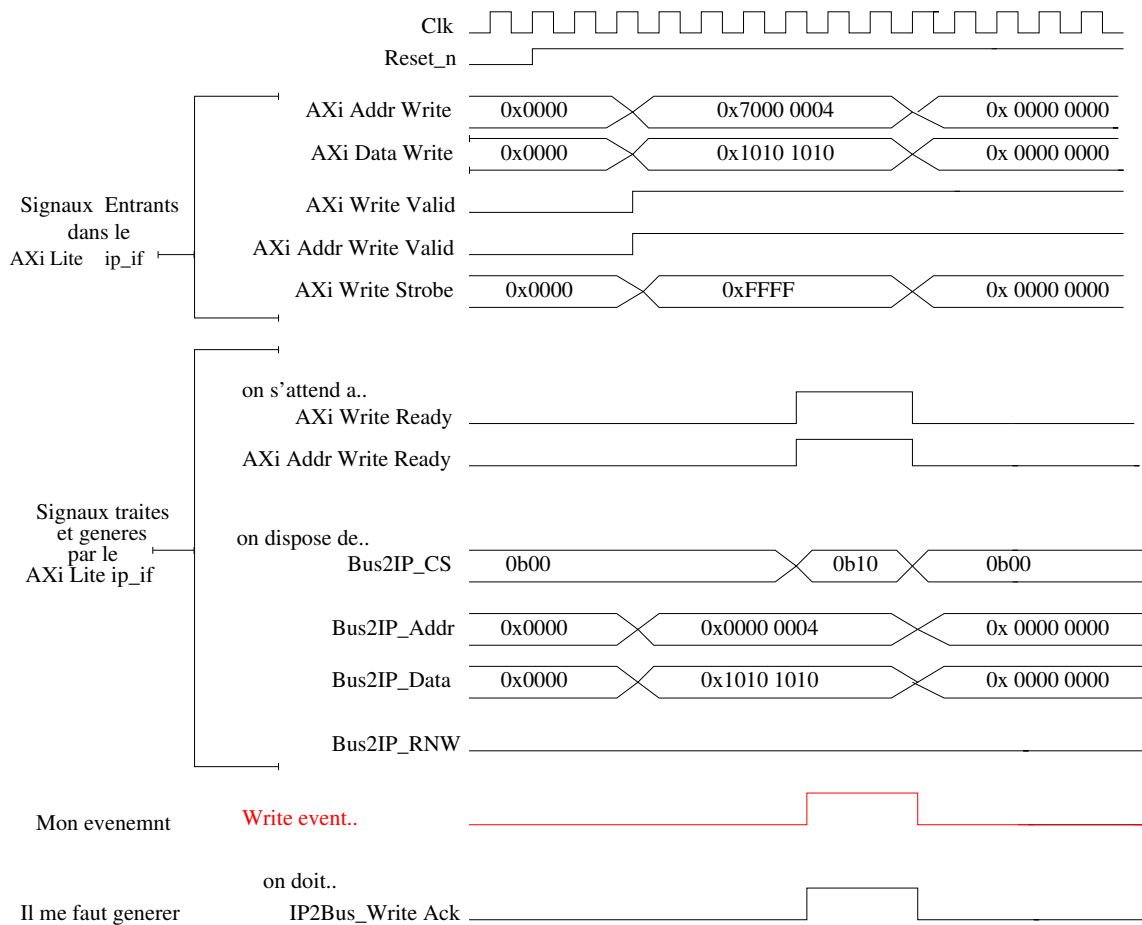


FIGURE 13 – AXi Lite Single Write Event

```
entity zynq_led is
generic (
    AXI_SIZE : integer := 32
);
port (
    -- AXi
    clk : in std_logic;
    rst : in std_logic;
    axi_addr : in std_logic_vector(AXI_SIZE-1 downto 0);
    axi_wrdata : in std_logic_vector(AXI_SIZE-1 downto 0);
    axi_rddata : out std_logic_vector(AXI_SIZE-1 downto 0);
    axi_cs : in std_logic_vector(1 downto 0);
    axi_rnw : in std_logic;
    axi_wrack : out std_logic;
    axi_rack : out std_logic;
    axi_error : out std_logic;
    -- appli
    leds : out std_logic_vector(6 downto 0)
);
end entity;
```

cL'en-

titée est la plus basique : on a l'ensemble des signaux du bus AXi Lite qui nous intéressent, et seulement une sortie leds en plus.

REG est une variable qui va pouvoir prendre les états voulus par l'application.  
Ses états sont dictés par la valeur passée en adresse sur le bus.  
Dès que l'on a une condition de WRITE ou de READ on acknowledge juste après pour éviter que le bus ne reste en suspend.  
Le bit d'error est maintenu a l'état bas.

```

1 AXLMUX: process(clk , rst)
begin
3     if rst = '1' then
        axi_wrack <= '0';
5         axi_rdack <= '0';
        axi_error <= '0';
7         read_s <= (others => '0');
        -- appli
9         leds_s <= (others => '0');
        elsif rising_edge(clk) then
11            read_s <= read_s;
            axi_error <= '0';
13            axi_rdack <= '0';
            axi_wrack <= '0';
            -- appli
15            leds_s <= leds_s;
            if axi_cs(0) = '1' and axi_rnw = '0' then
17                axi_wrack <= '1';
                case REG is
19                    when WRITE =>
                        leds_s <= axi_wrdata(6 downto 0);
21                    when IDLE =>
23                    when OTHERS =>
25                end case;
            end if;
27 end process AXLMUX;
29 end rtl;

```

programmes/axilitewrite.vhd

Ainsi vous pouvez voir ma condition "Write" ( depuis le CPU ).

```

1 AXLMUX: process(clk , rst)
begin
3     if rst = '1' then
        axi_wrack <= '0';
5         axi_rdack <= '0';
        axi_error <= '0';
7         read_s <= (others => '0');
        -- appli
9         leds_s <= (others => '0');
        elsif rising_edge(clk) then
11            read_s <= read_s;
            axi_error <= '0';
13            axi_rdack <= '0';
            axi_wrack <= '0';
            -- appli
15            leds_s <= leds_s;
            elsif axi_cs(0) = '1' and axi_rnw = '1' then
17                axi_rdack <= '1';
                case REG is
19                    when STATUS =>
                        read_s(6 downto 0) <= leds_s;
21                    when ID =>
                        read_s <= X"7e400000";
23                    when IDLE =>
25                    when OTHERS =>
27                end case;

```

```

27         end if;
        end if;
29 end process AXLMUX;
31 end rtl;

```

programmes/axiliteread.vhd

De même pour la condition "Read" ( depuis le CPU ).

**Architecture** NB : Sans respecter l'architecture générale suivante, ce code n'est pas valable.

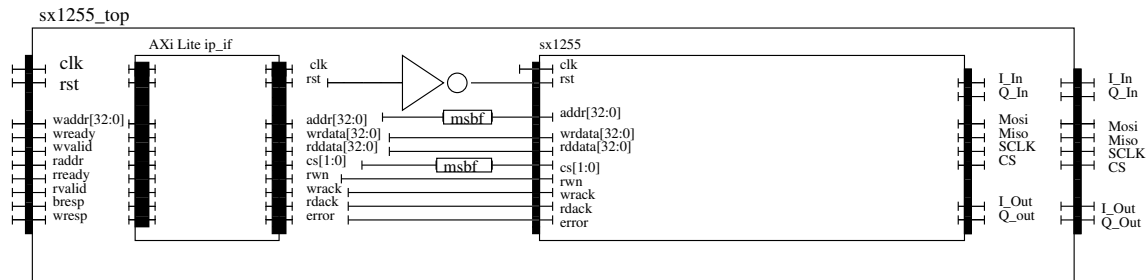


FIGURE 14 – connexion top et user\_logic

Le reset est remis en logique classique, les adresses et CS sont mises en MSBfirst, toutes ces opérations sont effectuées dans le user\_logic et mon bloc \_axi reçoit directement les signaux corrects.

Figure.

Dès que l'on crée un custom IP on a vu que l'on dispose d'un fichier top et d'un fichier user\_logic.vhd. Le fichier nomdemonmodule.top.vhd n'est jamais modifié mis à part l'addition de ports et de génériques spécifiques à l'application. Le fichier user\_logic.vhd instancie notre customip.vhd. Au sein du user\_logic.vhd je repasse le reset en logique habituelle, je passe Addr et le CS en MSB First. Les templates de lecture et d'écriture générés lors de la création peuvent être effacés.

## 8 AXi Lite Master

La variante Master permet au FPGA de passer lui aussi une adresse d'action sur le bus AXi.

### 8.1 Etude du protocole

**Timing diagram** Etudions le timing de ce nouveau protocole. inclure le graph.

### 8.2 Modèle AXi Lite Master VHDL

**Write event**

**Read event**

### 8.3

## 9 Mise en place d'une interface AXi Streaming

### 9.1 Etude du protocole

Le protocole AXi Streaming découle en plusieurs variantes, il est le plus rapide que nous puissions mettre en oeuvre sur le zynq.

Le principe est que, contrairement à l'AXi lite précédent, il n'y a pas de notion d'adressage, la communication a lieu entre deux entités fixes, et on a plus un simple transfert de données mais un certains nombres de "beats" de transfert - burst length -.

Le protocole est celui mis en place pour communiquer avec les HDL AXi DMA de Xilinx, donc il est important de valider son intégration pour pouvoir ajouter des applications dma.

**Timing** Dictées par le chronogramme suivant :

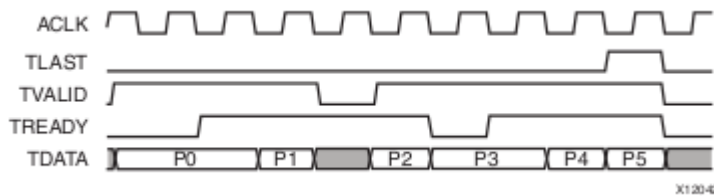


FIGURE 15 – AXi Streaming timing diagram

### 9.2 Exemple VHDL

## 10 Interrupt Request

Etude de la génération d'interruption FPGA -> CPU et CPU -> FPGA.

On peut dire qu'il est plus naturel d'imaginer une interruption générée par le FPGA pour le processeur. Pour l'instant je n'ai pas testé l'inverse mais on peut imaginer que cela ne soit pas beaucoup plus compliqué.

Contrairement au reste du chapitre, la génération de l'interruption se fait plutôt facilement.

Imaginons une application comme ceci :

On dispose de deux SW qui, si l'un ou l'autre est passé à l'état haut, générera une interruption vers le CPU.

Pour cela, générez un custom IP, par exemple interrupt\_top.vhd avec deux entrées : SW0 et SW1, et deux sorties : FLAG0 et FLAG1.

Liez par une simple logique FLAG0  $\leftarrow$  SW0 et FLAG1  $\leftarrow$  SW1.

Ajoutez l'IP comme on le fait d'habitude en important le .pao dans include existing peripheral.

Allez dans le fichier projet/pcores/interrupt\_top/data et dans le fichier .mpd ajoutez aux ports FLAG0 et FLAG1 les propriétés suivantes :

SIGIS = INTERRUPT, INTERRUPT\_SENSITIVITY = EDGE\_RISING, INTERRUPT\_PRIORITY = MEDIUM

Revenez dans XPS et dans projects cliquez Rescan User Repositories.

Dans le tabs PORTS vous devriez voir les propriétés INTERRUPT ajoutée aux sorties FLAG de l'IP.

Vous pouvez alors maintenant lier ces deux signaux dans le IRQF2P : interrupt request Fabric to Process Faites le en cliquant sur ce bouton dans le PORT Tab sous processing\_system7

## Interrupts dans le Zynq

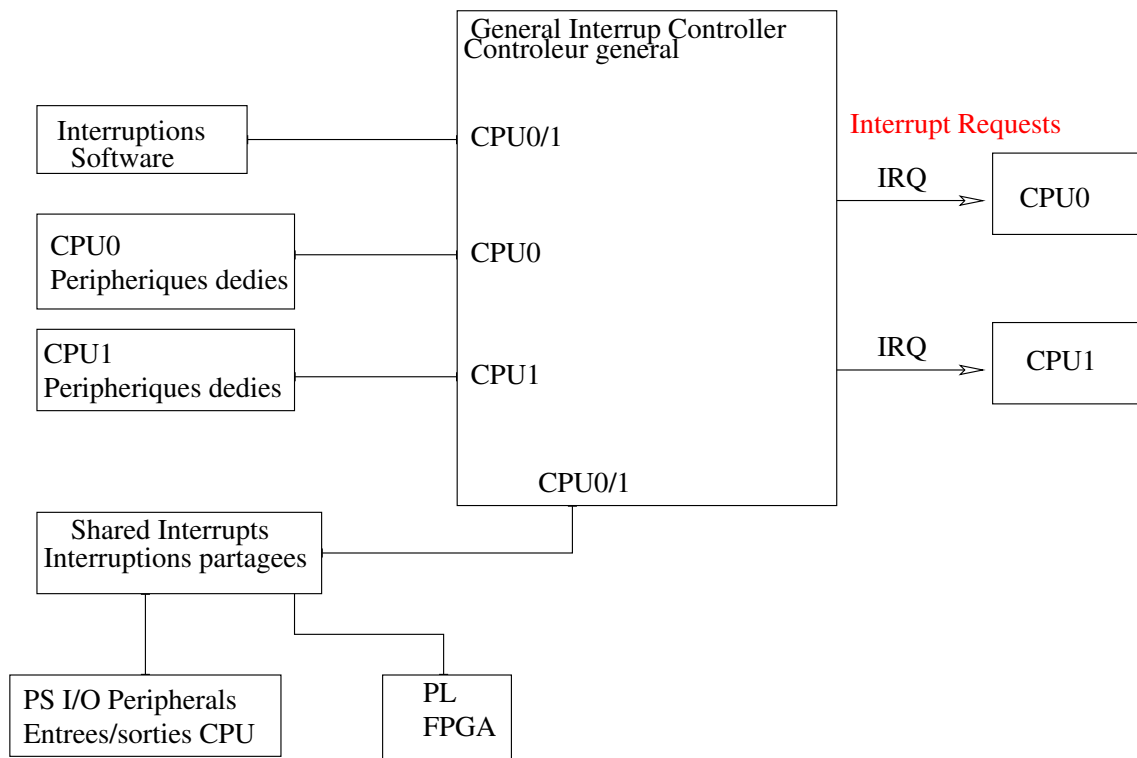


FIGURE 16 – Global Interrupt Controller

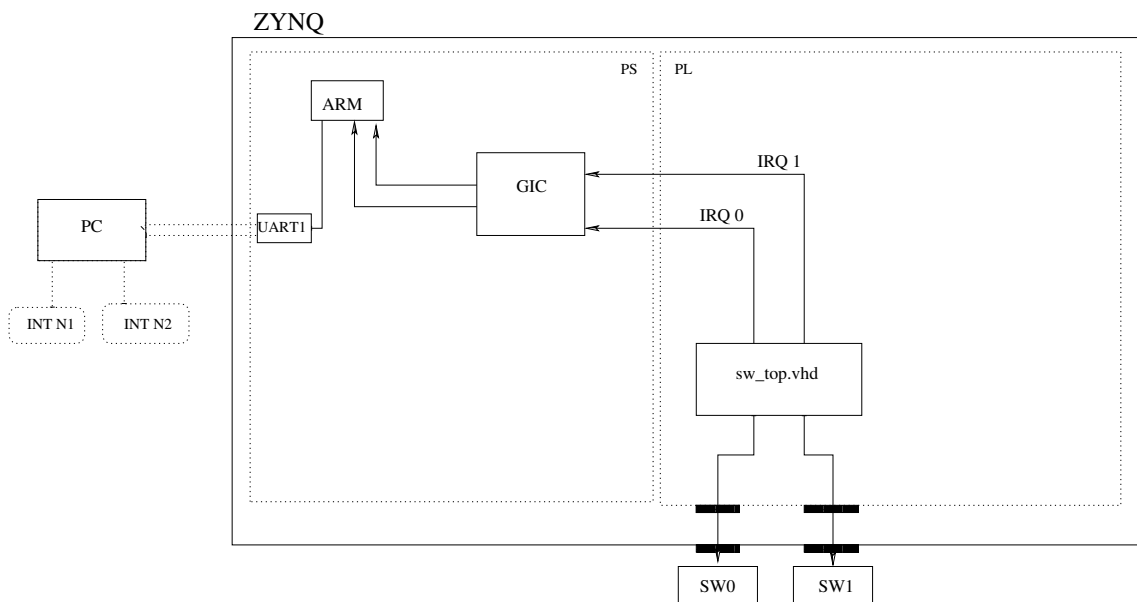


FIGURE 17 – application interrupt

Le système vient d'adresser des numéro de flags pour ces deux interruptions.

## 10.1 Partie software - Baremetal

# 11 Exemples d'application FPGA avancé

## 11.1 Zynq led

Nous allons créer l'application suivante, qui remplace notre application axi\_gpio du départ :

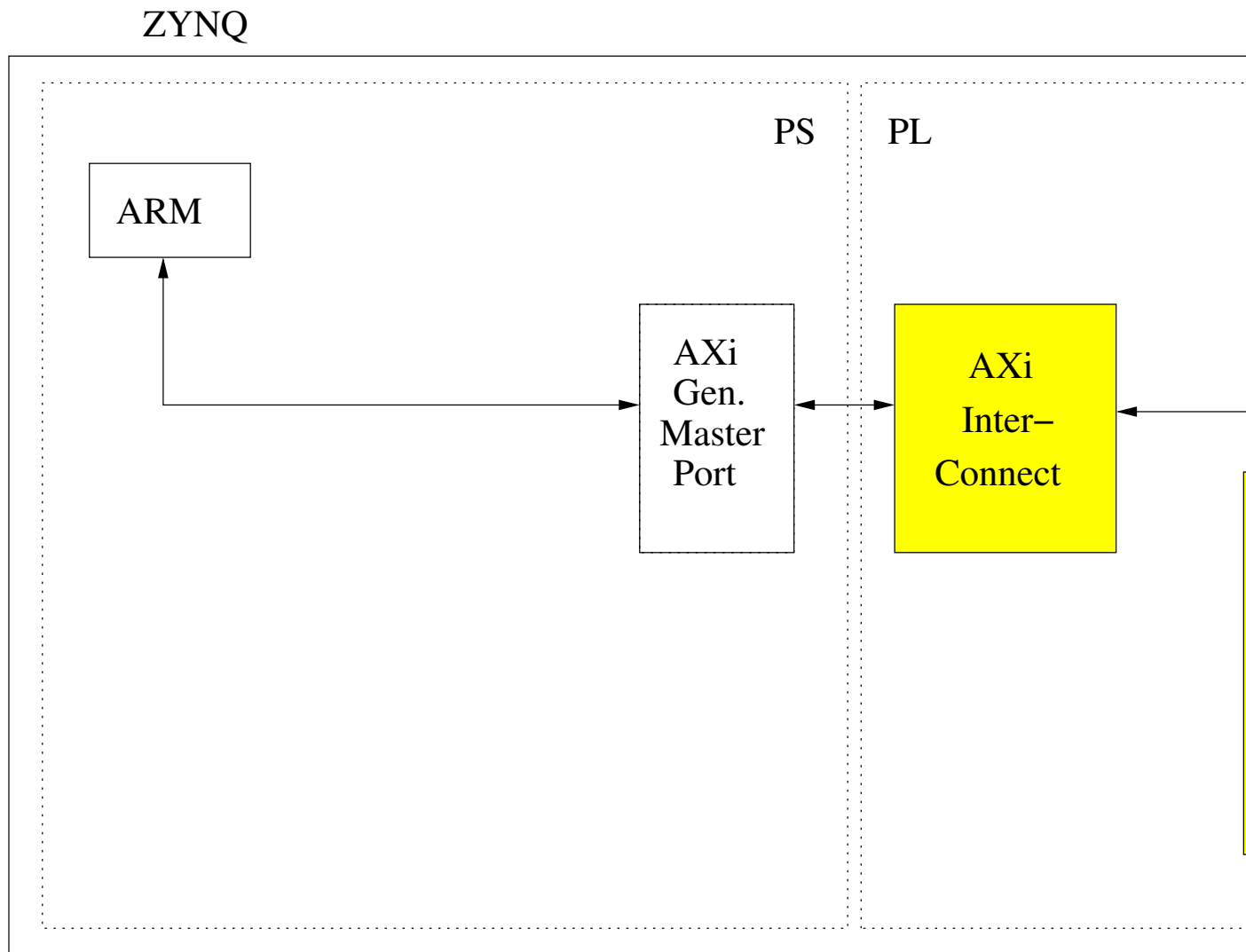


FIGURE 18 – architecture pour zynq led

Pour cela il nous faudra créer un custom ip, l'inclure selon la norme axi, étudier l'interconnexion sur le bus selon la norme axi slave, et nous vérifierons le bon fonctionnement

## 12 AXi DMA

### 12.1 Architecture Zedboard

Nous disposons d'un certains nombre de bloc proposés par Xilinx pour faire de la DMA. On peut citer :

Le bloc CDMA : Central DMA qui semble avoir pour rôle de stocker et de transférer des données depuis le CPU vers le FPGA et vice-versa mais sans avoir la possibilité d'y lier des entrées/sorties utilisateur.

Le bloc DMA : Dma engine qui est celui que nous utiliserons, qui prend logiquement une entrée streaming et une sortie streaming ( CPU -> FPGA — FPGA -> CPU ).

Le bloc VDMA : Video DMA, transfert streaming d'un vecteur à deux dimensions pour de la vidéo.

L'architecture du flux de données est également prendre plusieurs formes.

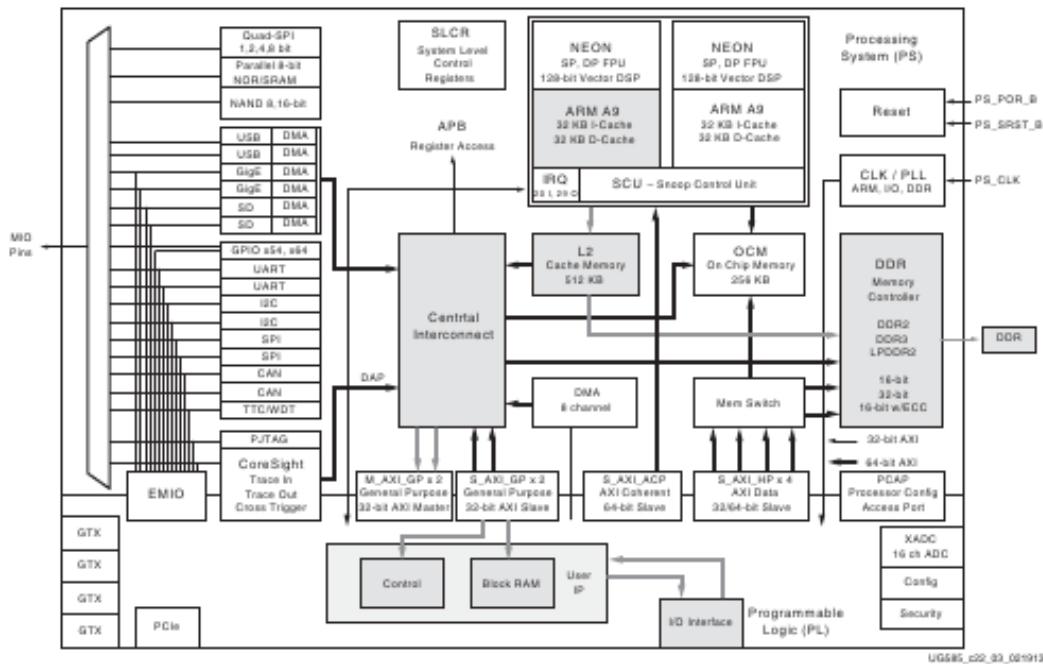


Figure 22-1: Example Cortex-A9 PL Data Movement Topology

FIGURE 19 – dma architecture sur le zynq

L'entité ACP ou Accelerator Coherency Port, va permettre la liaison du bloc DMA vers les mémoires cache : moins d'espace mémoire mais plus rapide. Le rôle du ACP est de garantir une cohérence entre le CPU et la zone mémoire L1Cache pendant que le transfert s'effectue vers la seconde zone L2Cache.

L'entité HP ou High Performance Port, va permettre la liaison du bloc vers les bloc de DDRRAM.

On peut également utiliser des interfaces AXi Slave/Master General Purpose Ports.

Extrait de la documentation officielle, comparaison des différentes méthodes possibles pour transférer des données sur le zynq et comparaison des débits envisageables : Figure 20.

Les High Performance Ports offrent le débit le plus important : 1.2Gbits/s et ce sont ceux que je vais utiliser. Il nous faut en spécifier un par liaison :

Si on veut transférer des données du CPU vers le FPGA en DMA il nous faut un HP.

Si on veut transférer des données du FPGA vers le CPU en DMA il nous faut un autre HP.

On peut en avoir 4 en même temps au maximum.

Le HP prendra en entrée et en sortie un flux de Memory Map qui sera convertie en streaming par le bloc DMA engine.



Table 22-8: Data Movement Method Comparison Summary

Method	Benefits	Drawbacks	Suggested Uses	Estimated Throughput
CPU Programmed I/O	<ul style="list-style-type: none"> <li>Simple Software</li> <li>Least PL Resources</li> <li>Simple PL Slaves</li> </ul>	<ul style="list-style-type: none"> <li>Lowest Throughput</li> </ul>	<ul style="list-style-type: none"> <li>Control Functions</li> </ul>	<25 MB/s
PS DMAC	<ul style="list-style-type: none"> <li>Least PL Resources</li> <li>Medium Throughput</li> <li>Multiple Channels</li> <li>Simple PL Slaves</li> </ul>	<ul style="list-style-type: none"> <li>Somewhat complex DMA programming</li> </ul>	<ul style="list-style-type: none"> <li>Limited PL Resource DMAs</li> </ul>	600 MB/s
PL AXI_HP DMA	<ul style="list-style-type: none"> <li>Highest Throughput</li> <li>Multiple Interfaces</li> <li>Command/Data FIFOs</li> </ul>	<ul style="list-style-type: none"> <li>OCM/DDR access only</li> <li>More complex PL Master design</li> </ul>	<ul style="list-style-type: none"> <li>High Performance DMA for large datasets</li> </ul>	1,200 MB/s (per interface)
PL AXI_ACP DMA	<ul style="list-style-type: none"> <li>Highest Throughput</li> <li>Lowest Latency</li> <li>Optional Cache Coherency</li> </ul>	<ul style="list-style-type: none"> <li>Large burst might cause cache thrashing</li> <li>Shares CPU Interconnect bandwidth</li> <li>More complex PL Master design</li> </ul>	<ul style="list-style-type: none"> <li>High Performance DMA for smaller, coherent datasets</li> <li>Medium granularity CPU offload</li> </ul>	1,200 MB/s
PL AXI_GP DMA	<ul style="list-style-type: none"> <li>Medium Throughput</li> </ul>	<ul style="list-style-type: none"> <li>More complex PL Master design</li> </ul>	<ul style="list-style-type: none"> <li>PL to PS Control Functions</li> <li>PS I/O Peripheral Access</li> </ul>	600 MB/s

FIGURE 20 – comparaison méthodes de transferts sur le zynq

## 12.2 Création d'une architecture DMA

Dans un nouveau projet sous XPS, nous pouvons procéder de deux manières : manuelle ou automatique.

Automatique : Le logiciel va lier les blocs entre eux convenablement et mettre en place l'interface BUS qui convient.

Ajoutez un bloc **AXi\_DMA\_Engine** et choisissez l'option High Performance port.

Dans ses paramètres choisissez : Disable Gather Engine, Enable S2MM - Enable MM2S qui permettent le transfert dans les deux sens, la taille de transfert et la taille des mots transférés.

Ajoutez un Custom IP sous la norme AXi streaming avec la taille de mots et la taille de transfert choisie.

Dans le bus interface reliez la partie S\_AXi du custom IP à M\_AXIS\_S2MM du dma engine.

Reliez la partie M\_AXi du custom IP à S\_AXIS\_MM2S du dma engine.

Utilisez le code source suivant :

Etude :

## 12.3 Création d'un custom IP avec transfert DMA

Il va nous falloir importer notre bloc décodant l'AXi Streaming que nous avons déjà vu.

## 13 Embedded Linux

Dans le contexte de l'exploitation du Zynq pour une application de traitement de signaux radiofréquences (*Software Defined Radio*), nous nous sommes inspirés de l'installation de GNU/Linux et outils associés au moyen de OpenEmbedded tel que décrit à <http://gnuradio.org/redmine/projects/gnuradio/wiki/Zynq>. L'objectif est d'obtenir une chaîne de compilation croisée, un noyau, les applications associées (*rootfs*) dans un environnement cohérent. Une originalité de la présence du FPGA tient

en la nécessité de configurer les périphériques accessibles aux processeurs : une configuration spécifique du FPGA est nécessaire en ce sens.

Les processeurs ARM sont souvent intégrés dans des plateformes plus complexes que sont les *System on Chips* (SOC) et la configuration des périphériques s'avère un problème complexe pour tenir compte de toutes les combinaisons possibles. Le choix s'est orienté vers un outil de configuration nommé *devicetree*.

## 14 Communication Linux - FPGA

### 14.1 Créer un bitstream à flasher depuis Linux

Linux démarre au travers de la première partition sur un *bitstream.BIT*. Or, pour reprogrammer notre FPGA, nous avons besoin d'un bitstream au format *.BIN*.

Pour obtenir un tel format, allez dans le repertoire de votre projet où se trouvent votre bitstream :

```
cd /repertoire_du_projet/nom_du_module.runs/impl_1/
```

 Conversion vers le format *.bin* :

```
bitgen nomdumodule_stub_routed.ncf nomdumodule_stub.bit nomdumodule_stub.pcf -b -w -g Binary :Yes
```

 Vous devriez disposer maintenant du fichier *nomdumodule\_stub.bin* .

Pour le flasher depuis Linux il nous faut invoquer le driver */dev/xdevcfg*

```
mknod /dev/xdevcfg c 259 0
```

```
chmod 666 /dev/xdevcfg
```

```
cat my_.bin > /dev/xdevcfg
```

 Et vous devriez voir la LED12(DONE) du fpga d'abord s'éteindre puis se rallumer si tout va bien.

### 14.2 Linux - IP user interface

Utilisation de mes 3 IP qui s'interfacent sur le bus AXi et permettent le controle et l'utilisation des boutons, leds, switches du fpga.

Pour pouvoir les utiliser, il vous faut flasher depuis Linux le bitstream contenant ces IP.

**zynqled** permet le contrôle des 8x Leds du fpga

**zynqbtn** permet de lire quel bouton est est presse dans le fpga

**zynqsw** permet de lire l'état des 8x Sws du fpga

### 14.3 Ajout de l'ip dans le device tree

Il nous faut ajouter dans notre devicetree, la nouvelle plage d'adresse correspondant à notre IP. Par exemple, mon ip zynqled est défini sur l'intervalle [0x74E0 0000; 0x74E0 ffff].

Exemple d'implentation dans le devicetree pour cet intervalle et cet ip :

### 14.4 Communication avec l'ip depuis Linux

**via /dev/mem** Une première approche consiste à utiliser le driver */dev/mem*, se mapper sur la plage d'adresse correspondant à

<http://gnuradio.org/redmine/projects/gnuradio/wiki/Zynq>

## 15 Conclusion

Si quelque chose n'est pas clair, devenu obsolete, vous semble induit d'erreur ou est totalement faux, n'hésitez pas a nous contacter : .

```

ps7-usb@e0003000 {
    clocks = <0x3 0x1d>;
    compatible = "invalid";
    reg = <0xe0003000 0x1000>;
    interrupt-parent = <0x2>;
    interrupts = <0x0 0x2c 0x4>;
    dr_mode = "host";
    phy_type = "ulpi";
    xlnx,usb-reset = <0xffffffff>;
};

ps7-xadc@f8007100 {
    clocks = <0x3 0xc>;
    compatible = "xlnx,ps7-xadc-1.00,a";
    reg = <0xf8007100 0x20>;
    interrupt-parent = <0x2>;
    interrupts = <0x0 0x7 0x4>;
};

zynqled@7E400000 {
    compatible = "zynqled";
    reg = <0x7e400000 0x10000>;
};

```

FIGURE 21 – exemple d'implémentation dans le devicetree