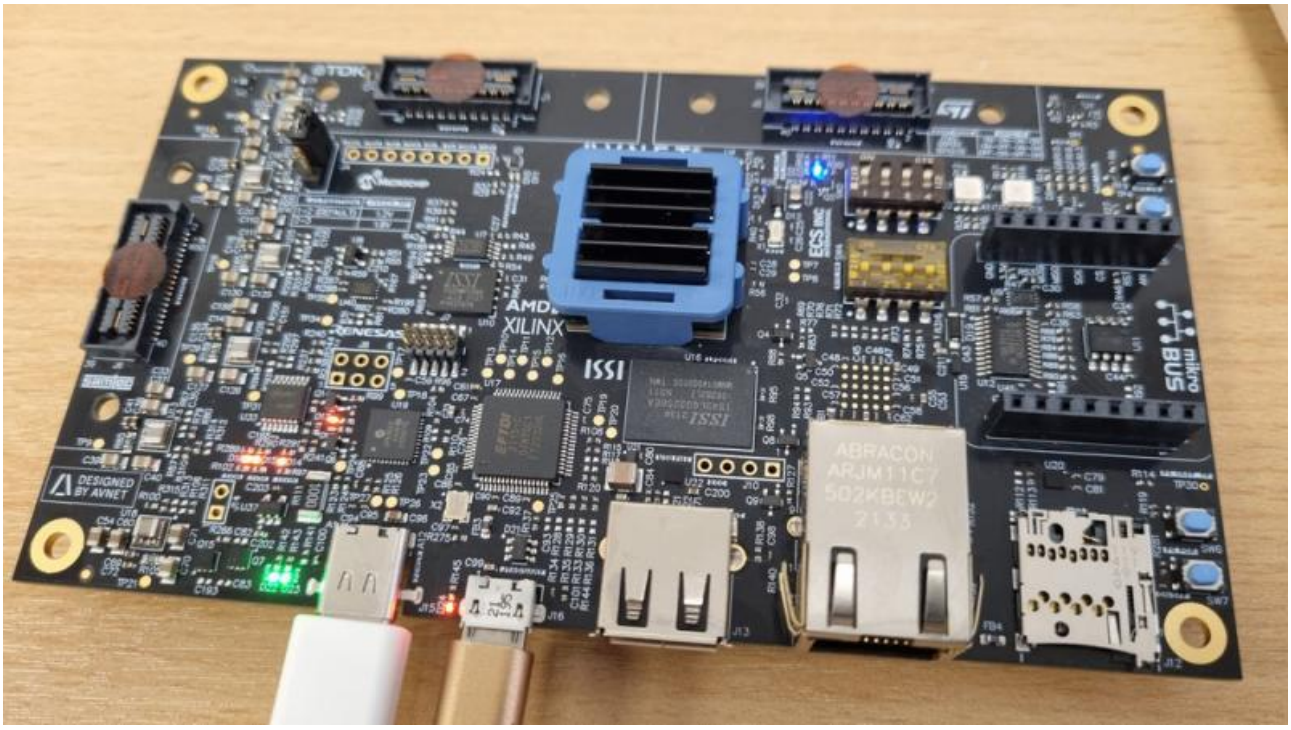
BORDEAUX INP Enseirb-Matmeca

Rapport Projet avancé :

Codesign sur carte Avnet ZuBoard avec FPGA MPSoC 1CG



Encadrant :

KADIONIK Patrice

Auteurs :

RIHANE Ossama

BEL HADDAD El Mehdi

GHOUZALI El Mehdi

DEQUIRET William

Sommaire

1. Introduction 3
2. Les GPIOs 5
3. Communication avec les capteurs de la carte ZuBoard 7
3.1. Lecture de température par I2C : 7
3.2. Lecture de température par I2C : 10
4. Mise en place du projet Petalinux 11
5. Mise en place d'une distribution personnalisée avec Xilinx Open Source Linux 17
6. Conclusion 21
7. Annexe 22
1. Code source .c pour le test GPIO 22
2. Code source .c pour le test de température I2C 24
3. Code source .c pour le test de température SPI et I2C avec petalinux 26
4. Références 30

Table de figures

Figure 1 - Carte de développement ZuBoard 3
Figure 2 - Carte de développement ZedBoard 4
Figure 3 : Interfaces de la ZuBoard 4
Figure 4 - Boutons poussoir et Switchs 5
Figure 5 - Design bloc à configurer 5
Figure 6 - Capteur de température STTS22HTR 7
Figure 7 - Table des registres du capteur de température 8
Figure 8 - Système Zynq MPSoC comportant le bloc de température I2C 8
Figure 9 - Mesures de température 10
Figure 10 - Petalinux 11
Figure 11 - activation de l'option 'User mode SPI device driver support' 12
Figure 12 - configuration du rootfs pour inclure les outils I2C 13
Figure 13 - vérification de l'existence de spi et i2c à l'aide de la commande ls /dev/ 14
Figure 14 - exécution de la commande sudo i2cdetect -y -r 3 15
Figure 15 - exécution de la commande sudo i2cget 3 0x3f 0x01 15
Figure 16 - exécution du code de test de spi et i2c sur la carte. 16
Figure 17 - Configuration d'u-boot 18
Figure 18 - Distribution Xilinx Linux sur la carte cible 19
Figure 19 - test hello world avec la distribution personnalisée 19
Figure 20 - Test SPI et I2C pour les données de température 20

1. Introduction

L'objectif de ce projet était de faire du codesign sur carte de développement Avenet ZuBoard.

Cette carte de développement, sortie en 2023, donc très récente, n'a pas encore d'abondance de tutoriel, et c'était donc l'occasion de découvrir et faire fonctionner ce matériel nouveau.

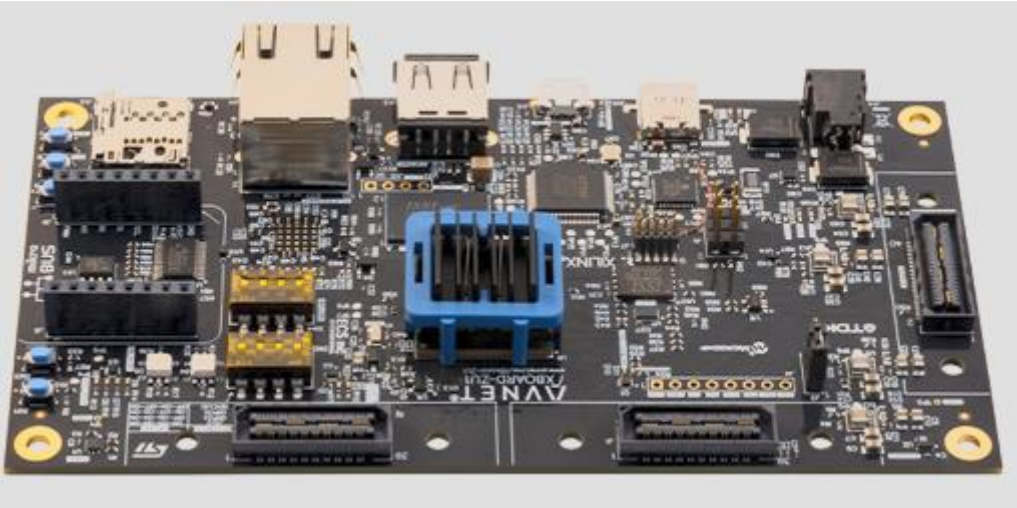


Figure 1 - Carte de développement ZuBoard

Heureusement, elle reste par bien des aspects tout de même assez comparables, dans la façon de la configurer, à la ZedBoard, une carte de développement bien plus documentée car plus ancienne, équipé d'un FPGA Zynq-7020 plutôt que d'un MPSoC-1CG pour la ZuBoard. Cela a permis de simplifier l'apprentissage de l'API Xilinx, car la ZedBoard possède de nombreux tutoriel, que nous avons adapté à la ZuBoard.

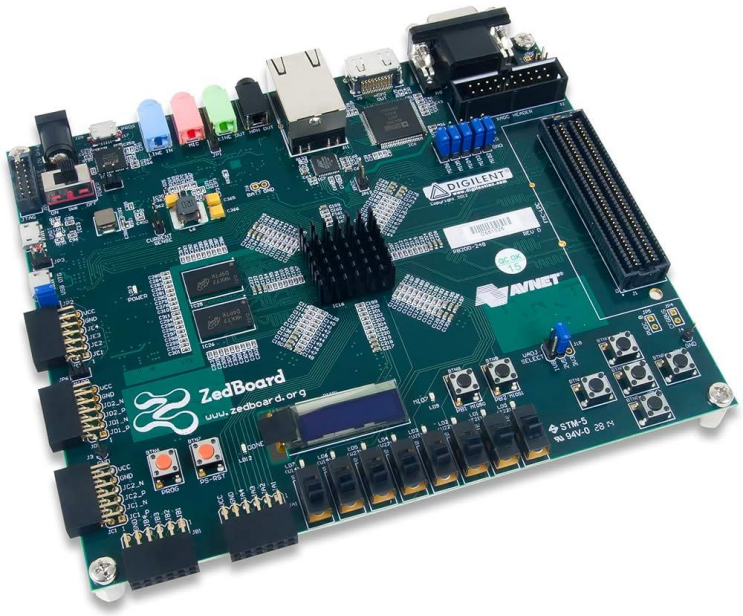


Figure 2 - Carte de développement ZedBoard

Cela a permis de simplifier l'apprentissage de l'API Xilinx, car la ZedBoard possède de nombreux tutoriel, que nous avons adapté à la ZuBoard. La Zuboard comprend de nombreuses interfaces, comme le montre la figure ci-dessous.

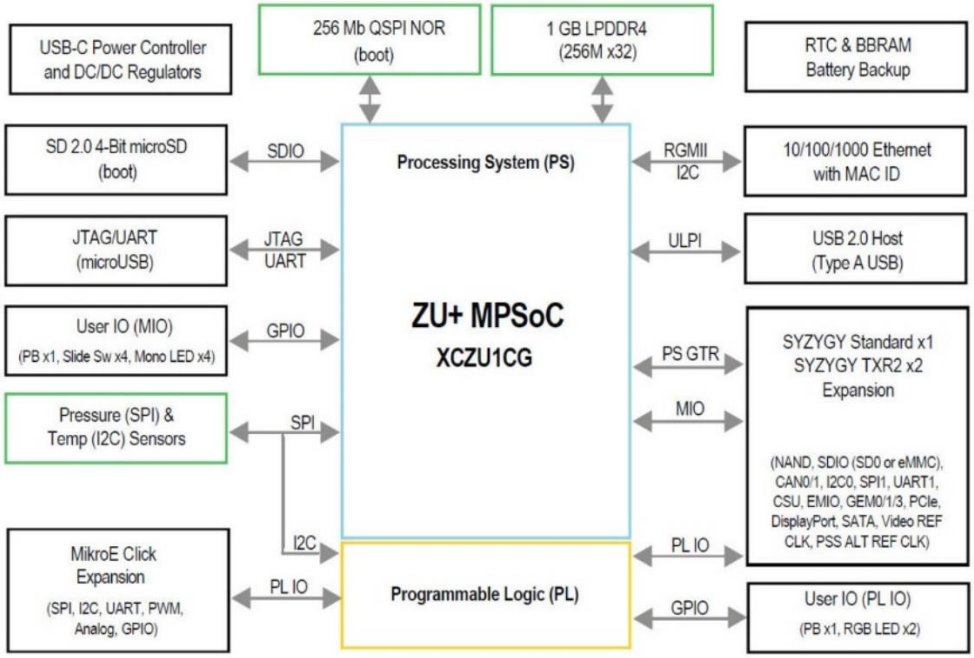


Figure 3 : Interfaces de la ZuBoard

Les interfaces étant nombreuses, nous nous sommes concentrés sur celles liées au Processing System (PS) plutôt que celles liées à la Logique programmable (PL).

Nous avons décidé de nous intéresser en premier lieu aux GPIOs, qui nous semblaient être le point de départ le plus logique.

2. Les GPIOs

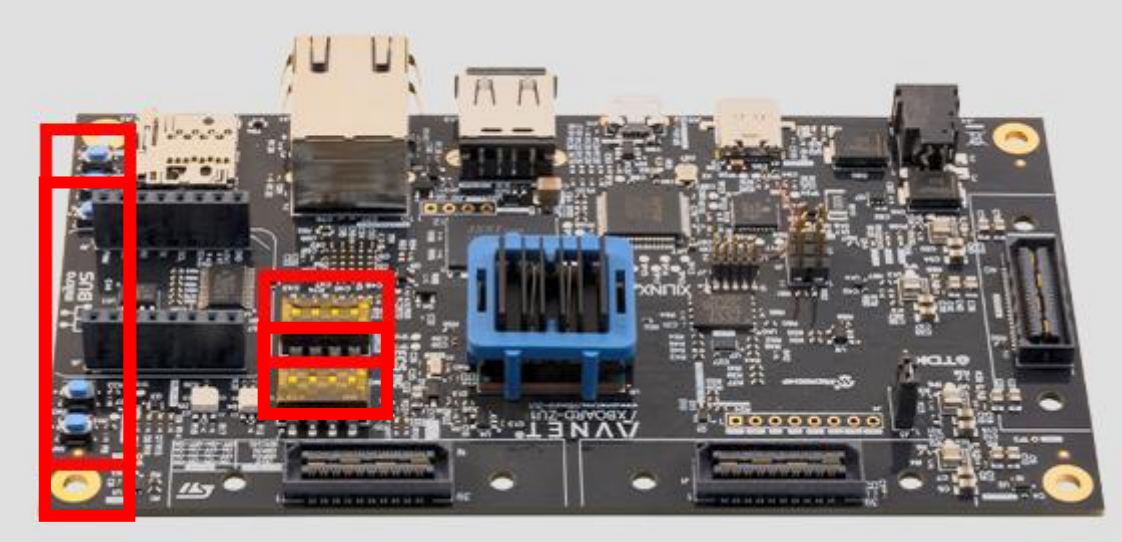


Figure 4 - Boutons poussoir et Switchs

La carte de développement ZuBoard est équipé de différents éléments permettant de tester les GPIOs (General Purpose Input/Output) : on peut notamment citer les LEDs, les Boutons poussoirs ainsi que les Switchs.

Faire fonctionner ces GPIOs est un bon point de départ pour prendre en main la carte de développement

Pour commencer, nous avons dû définir un design nous permettant d'utiliser les GPIOs. Pour cela, nous avons créé un design sous Vivado, que nous avons configuré.

Figure 5 - Design bloc à configurer

Puis, nous avons dû nous familiariser avec l'API (Application Program Interface) de Xilinx.

Il a tout d'abord fallu configurer la carte de manière à pouvoir rendre accessibles les gpios. Dans l'API Xilinx, cela passe par la fonction **XGpioPs_LookupConfig(XPAR_XGPIOPS_0_DEVICE_ID) ;**

Puis on initialise la configuration au moyen de la fonction : **XGpioPs_CfgInitialize(&Gpio,Config_Ptr,Config_Ptr->BaseAddr);**

S'en suit ensuite la configuration des GPIOs. Ceux-ci peuvent être soit des entrées, soit des sorties. On définit des macros permettant de repérer un pin par une appellation. Par exemple, le pin 7 est repéré par l'appellation LED1, le pin 32 est repéré par BP (Bouton Poussoir), et le pin 44 est repéré par SW1 (Switch 1). Nous configurons donc cela en choisissant la direction des GPIOs et en activant ou non le pin en sortie. Cela se fait au moyen des fonctions **XGpioPs_SetDirectionPin(…)** et **XGpioPs_SetOutputEnablePin(…)** :

- Par exemple, pour configurer la LED1 en sortie, on utilise les fonctions :
  XGpioPs_SetDirectionPin(&Gpio, LED1, 1);
  et
  XGpioPs_SetOutputEnablePin(&Gpio, LED1,1);

- Pour configurer le Bouton Poussoir en entrée, on utilise les fonctions :
  XGpioPs_SetDirectionPin(&Gpio, PB, 0);
  et
  XGpioPs_SetOutputEnablePin(&Gpio, PB,0);

Enfin, on peut venir lire ou écrire sur les GPIOs, selon qu'ils soient en entrée ou en sortie, en utilisant les fonctions : **XGpioPs_ReadPin(…)** et **XGpioPs_WritePin(…)** :

- Par exemple, pour le bouton poussoir : **PB_value = XGpioPs_ReadPin(&Gpio, PB);**

- Pour la LED1 : **XGpioPs_WritePin(&Gpio, LED1, value1 && PB_value);** (avec value1 la valeur du switch 1 (**value1 = XGpioPs_ReadPin(&Gpio, SW1);**))

Ne reste alors plus qu'à placer ces fonctions de lecture et d'écriture dans une boucle infini avec un sleep de 100ms de manière à pouvoir tester le bon fonctionnement de tous les GPIOs. Le code source utilisé pour tester cette partie se trouve en annexe (1. Code source .c pour le test GPIO).

Nous passons à l'étape suivante : faire fonctionner l'I2C et le SPI.

3. Communication avec les capteurs de la carte ZuBoard

3.1. Lecture de température par I2C :

Dans cette partie, nous allons utiliser le capteur STTS22HTR disponible sur la carte ZUBoard pour calculer la température via le protocole I2C.

Figure 6 - Capteur de température STTS22HTR

Ce capteur a une interface qui comporte sept registres accessibles pour l'échange de données via le protocole I2C , parmi lesquels quatre registres nous seront utiles :

- **WHO AM I** : Il comporte une valeur spécifique(0xA0 dans notre cas) qui permet de vérifier qu'on communique avec le périphérique attendu.
- **CTRL** : Ce registre est utilisé pour configurer et contrôler le comportement du périphérique I2C.Dans notre cas , on l'utilise pour activer l'incrémentation automatique des adresses et le mode de fonctionnement libre (« Free run ») afin de permettre au capteur d'effectuer des mesures de manière continue sans interruption externe.
- **TEMP_L_OUT** et **TEMP_H_OUT** : Ces deux registres permettent de diviser la sortie de température en deux Bytes :haut (high) et bas (low) qui forment la valeur complète de la température mesurée.

Table 12. **Register map**

Addr	Type(1)	Name	7	6	5	4	3	2	1	0	Default
01h	RO	WHOAMI	whoami7	whoami6	whoami5	whoami4	whoami3	whoami2	whoami1	whoami0	A0h
02h	RW	TEMP_H_LIMIT	THL7	THL6	THL5	THL4	THL3	THL2	THL1	THL0	00h
03h	RW	TEMP_L_LIMIT	TLL7	TLL6	TLL5	TLL4	TLL3	TLL2	TLL1	TLL0	00h
04h	RW	CTRL	LOW_ODR _START	BDU	AVG1	AVG0	IF_ADD _INC	FREERUN	TIME_OUT _DIS	ONE_ SHOT	00h
05h	RO	STATUS	0	0	0	0	0	UNDER_THL	OVER_THH	BUSY	output
06h	RO	TEMP_L_OUT	T7	T6	T5	T4	T3	T2	T1	T0	output
07h	RO	TEMP_H_OUT	T15	T14	T13	T12	T11	T10	T9	T8	output

1. *RW designates a read/write register. RO designates a read-only register*

Figure 7 - Table des registres du capteur de température

Notre design du système **Zynq MPSoC** étant déjà établi, on y ajoute le bloc IP de température I2C de la PL à partir de l'onglet « Boards » sous Vivado. En exécutant l'automatisation de la connexion, les interconnexions AXI nécessaires et la structure du reset sont créées. Ensuite, on crée le **HDL Wrapper** et on génère le **Bitstream** pour enfin exporter le projet et l'importer dans un nouveau projet Vitis pour y ajouter un code en langage C servant à piloter le capteur.

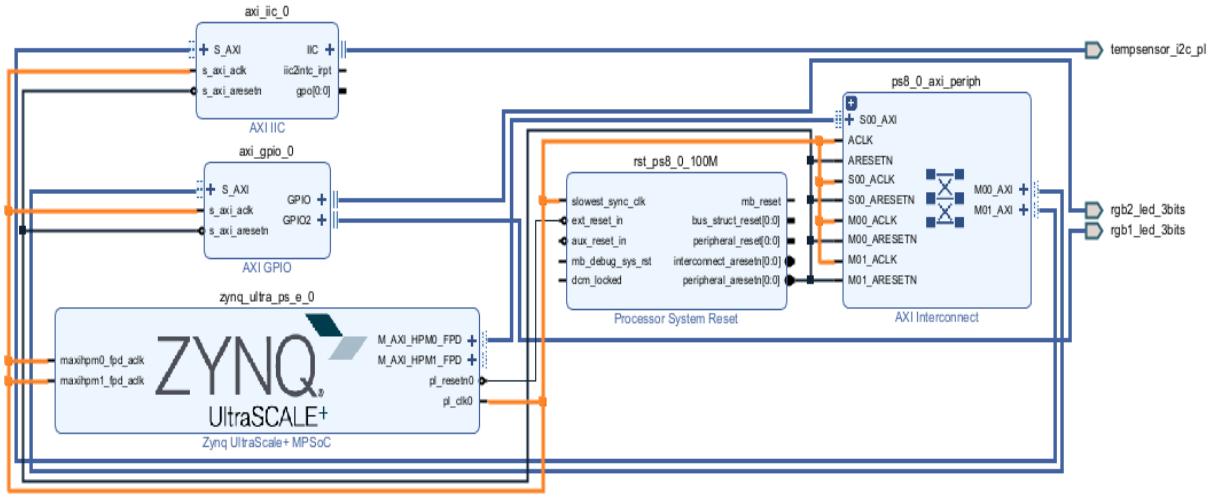


Figure 8 - Système Zynq MPSoC comportant le bloc de température I2C

Pour accéder au périphérique I2C, on utilise la bibliothèque **XIIC.h** disponible sous la BSP de la ZUBoard.

Ensuite, on initialise le périphérique I2C et on envoie une séquence I2C avec la valeur 0x01 pour vérifier la présence du capteur de température en utilisant les deux fonctions XIIC_Send et XIIC_Recv pour envoyer et recevoir des Bytes depuis les registres du capteur. Les buffers SendBuffer et RecvBuffer sont de taille 16bits et servent à stocker les données à envoyer et recevoir.

```
SendBuffer[0] = 0x01;
XIic_Send(iic.BaseAddress,IIC_SLAVE_ADDR,(u8 *)&SendBuffer, 1,XIIC_REPEATED_START);
XIic_Recv(iic.BaseAddress,IIC_SLAVE_ADDR,(u8 *)&RecvBuffer, 1,XIIC_STOP);

if(RecvBuffer[0]==WHOAMI){
    printf("Temp Sensor Detected\n\r");
}
else{
    printf("Temp Sensor NOT Detected\n\r");
    return 0;
}
```

Une fois le périphérique détecté, on envoie d'abord la valeur 0x04 pour indiquer qu'on échange avec le registre de contrôle puis on écrit dans ce registre la valeur 0x0C(0b00001100) qui permet d'activer l'incrémentation des adresses et le mode de fonctionnement libre (Voir Figure 7).

```
SendBuffer[0] = 0x04;
SendBuffer[1] = 0x0C;
XIic_Send(iic.BaseAddress,IIC_SLAVE_ADDR,(u8 *)&SendBuffer, 2,XIIC_STOP);
```

Ensuite, on lit la valeur stockée dans le registre TEMP_L_OUT(0x06) pour récupérer la partie basse de la température qu'on combine avec la partie haute pour obtenir la valeur complète sur 16bits.Cette valeur est ensuite convertie en virgule flottante puis divisée par 100 pour prendre en compte la représentation décimale de la température.

```
SendBuffer[0] = 0x06;
XIic_Send(iic.BaseAddress,IIC_SLAVE_ADDR,(u8 *)&SendBuffer, 1,XIIC_REPEATED_START);
XIic_Recv(iic.BaseAddress,IIC_SLAVE_ADDR,(u8 *)&RecvBuffer, sizeof(RecvBuffer),XIIC_STOP);
result = RecvBuffer[1] << 8 | RecvBuffer[0];
temp = (float) result / 100;
printf("Temperature is %f \n\r",temp);
```

L'exécution du code montre des valeurs de température autour de 35°C qui représente une valeur supérieure à la température ambiante de la salle P159 puisque le capteur de température se trouve à proximité du processeur de la carte.

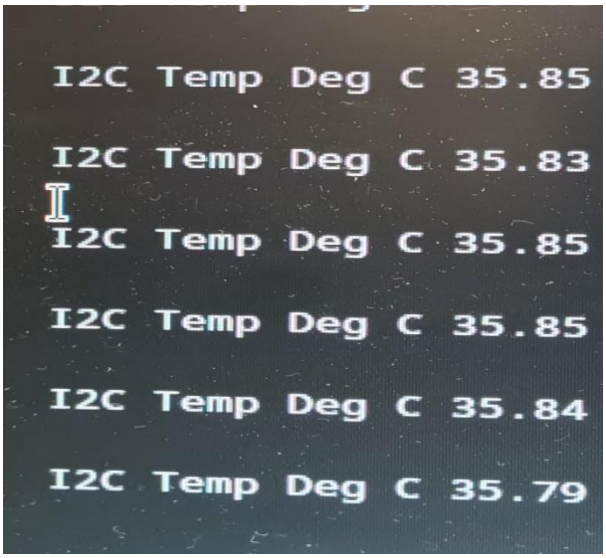


Figure 9 - Mesures de température

3.2. Lecture de température par I2C :

En utilisant deux versions de code différentes, la première étant la nôtre et la deuxième étant celle fournie par le site ADIUVOENGINEERING (Voir Références) qui implémente le même principe d'échange que le code précédent pour le capteur de température, nous n'avons pas réussi à faire fonctionner le capteur de pression par SPI. Cela nous mène à conclure qu'il y a un défaut matériel ou un problème de connexion au bus SPI qui entraîne des lectures incorrectes.

Pour la lecture de température via SPI, la configuration du pilote « spidev » via le **Device tree** est nécessaire pour faire fonctionner le capteur de température par SPI. Cette configuration sera traitée dans la partie qui suit où nous allons installer et configurer Petalinux sur la plateforme ZUBoard.

4. Mise en place du projet Petalinux

La première étape dans cette partie est d'installer Petalinux qui va nous aider à utiliser les capteurs du protocole SPI.

En effet, PetaLinux est un environnement de développement open-source développé par Xilinx, et conçu pour simplifier la création et la personnalisation de systèmes d'exploitation Linux embarqués destinés aux systèmes sur puce (SoC) FPGA de Xilinx. Il offre un ensemble d'outils facilitant la configuration, la construction et la gestion des images Linux pour les plateformes matérielles basées sur les FPGA de Xilinx. Il simplifie également le processus de développement en fournissant des outils de gestion de configuration tel que l'outil graphique qui permet de configurer facilement les options du noyau Linux et d'autres composants du système, des scripts de démarrage pré-configurés et des fonctionnalités avancées facilitant ainsi la création d'applications embarquées performantes et fiables.

Figure 10 - Petalinux

D'abord, il faut télécharger la version Petalinux 2022.2 à partir du site officiel de Xilinx pour qu'elle soit compatible avec les versions 2022.2 de Vivado et Vitis utilisées précédemment.

Dès que Petalinux 2022.2 est installé, il faut charger les variables d'environnement spécifiques à Petalinux dans la session shell utilisée, afin de pouvoir exécuter les commandes nécessaires pour créer notre projet personnalisé Petainux et effectuer les configurations requises. Pour ce faire, il est recommandé d'utiliser le script settings.sh, qui définit diverses variables d'environnement nécessaires pour configurer l'environnement de développement Petalinux, telles que les chemins d'accès aux outils de compilation, les options de configuration, et d'autres paramètres nécessaires à la construction de l'image Linux. Ainsi, pour charger ces commandes, il suffit de se placer dans le répertoire de Petalinux 2022.2 et d'exécuter la commande suivante : **source settings.sh**

Par conséquent, il devient possible d'utiliser les commandes offertes par petalinux pour la suite.

La première commande de petalinux à utiliser sert à créer le projet ciblant la carte ZuBoard :

petalinux-create -t project -n nom_du_projet --template zynqMP

Ensuite, il faut configurer ce projet en utilisant le fichier XSA provenant de du projet Vivado de la partie précédente à l'aide de la commande suivante :

petalinux-config --get-hw-description=/chemin/vers/projet/vivado

La prochaine étape consiste à configurer le noyau pour inclure la prise en charge du pilote de périphérique SPI en mode utilisateur, pour ce faire il faut d'abord lancer la commande suivante : **petalinux-config -c kernel**. Un outil graphique apparaîtra qui a pour rôle de configurer facilement les options du noyau Linux. Il suffit alors d'activer l'option 'User mode SPI device driver support' :

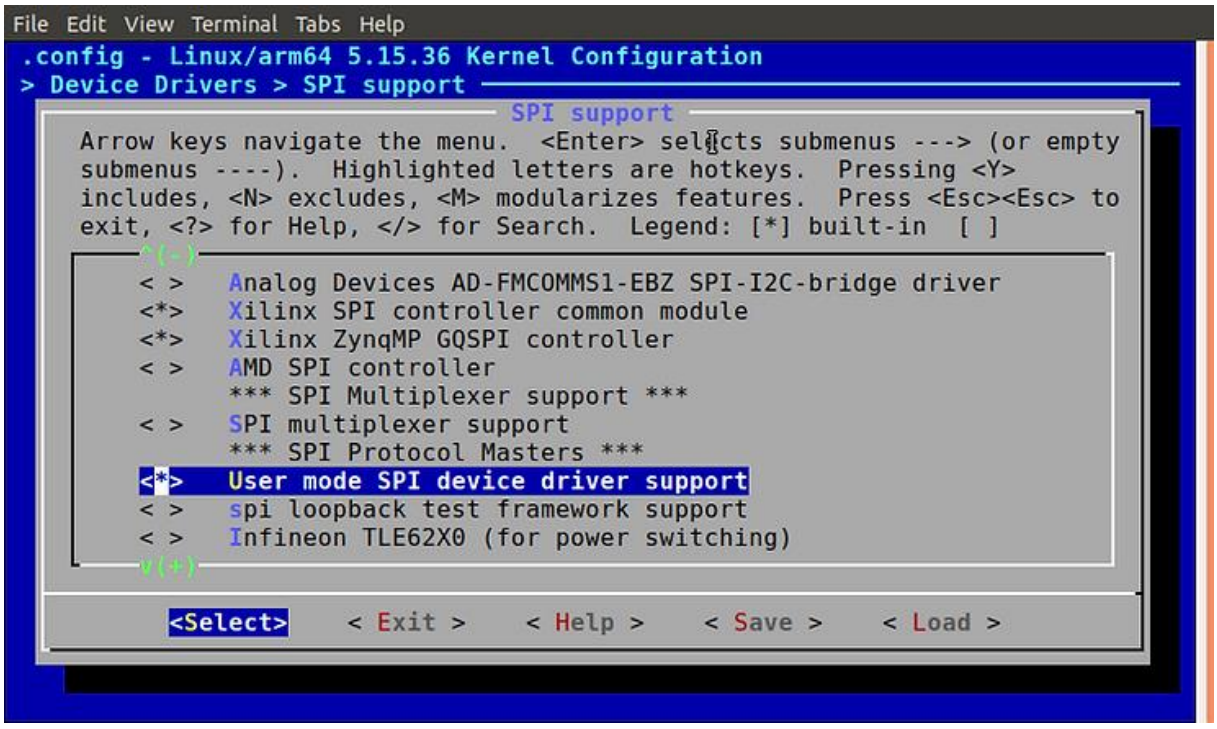


Figure 11 - activation de l'option 'User mode SPI device driver support'

Cette étape a permis d'assurer la communication avec des périphériques SPI à partir de l'espace utilisateur du linux embarqué construit par Petalinux.

L'étape suivante consiste à configurer le système de fichiers racine (rootfs) pour inclure les outils I2C. Pour cela, on commence par lancer la commande suivante : **petalinux-config -c rootfs**. Un outil graphique apparaîtra, permettant de personnaliser les composants et les options de la racine du système de fichiers (rootfs). Il suffit alors d'activer les options qui sont affichées dans la figure suivante :

Figure 12 - configuration du rootfs pour inclure les outils I2C

Cette étape garantit que notre carte dispose des ressources logicielles nécessaires pour travailler avec des périphériques I2C, assurant ainsi que les outils I2C seront présents et disponibles dans l'environnement d'exécution de notre carte.

L'étape suivante consiste à modifier le fichier device tree **system-conf.dtsi** pour activer le pilote « spidev » sur les deux ports SPI et activer le périphérique AXI IIC, cette étape permet de configurer notre carte afin de prendre en charge les communications SPI sur les deux ports et d'activer l'interface I2C sur le bus AXI, ce qui permet une communication efficace avec des périphériques connectés via ces interfaces. Cela permet à notre linux embarqué construit par Petalinux de reconnaître, configurer et interagir de manière efficace avec les périphériques connectés à ces interfaces sur la carte.

Par conséquent, la configuration des différents outils est terminée et la prochaine étape consiste à compiler le système linux embarqué personnalisé construit par Petalinux à l'aide de la commande : **petalinux-build**.

Cette commande est utilisée pour compiler le noyau Linux configuré, générer l'image du noyau ainsi que le système de fichiers racine (rootfs). Les détails de ce que cette commande fait sont les suivants :

- **Compilation du Noyau :** La commande **petalinux-build** compile le noyau Linux configuré pour notre carte ZuBoard. Cela inclut la configuration du noyau pour prendre en charge les périphériques et les fonctionnalités que nous avons déjà sélectionnés, comme les ports SPI, I2C.

- **Génération du Système de Fichiers Racine (RootFS) :** La commande **petalinux-build** génère également le système de fichiers racine (RootFS) qui sera utilisé par notre linux embarqué.
- **Configuration des Périphériques :** La commande **petalinux-build** prend en compte toutes les configurations que nous avons spécifiées dans les étapes précédentes, y compris celles du fichier device tree.
- **Génération de l'Image du Boot :** La commande **petalinux-build** génère l'image du boot qui sera chargée sur notre carte. Cette image inclut le noyau Linux, le système de fichiers racine(rootfs), et d'autres composants nécessaires au démarrage de notre linux embarqué construit par Petalinux.

Ensuite il faut utiliser la commande **petalinux-package** pour créer un package de démarrage. Ce package va être ensuite transféré sur la carte SD.

Une fois que les fichiers **boot.bin** et **image.ub** et **rootfs.tar.gz** sont obtenus, il faut les copier sur une carte SD et insérer cette carte dans notre carte ZuBoard pour la démarrer avec l'image linux construite par PetaLinux.

Ensuite, il faut connecter notre carte ZuBoard au réseau du PC via son connecteur Ethernet, par conséquent il faut configurer son adresse IP à l'aide des commandes **ifconfig** et **ip addr add**.

Ensuite, il faut vérifier si l'image PetaLinux contient tout ce dont nous avons besoin, en vérifiant que tous les périphériques I2C et SPI sont reconnus avec les bons pilotes, donc dans le terminal, il faut exécuter la commande **ls /dev** pour vérifier l'existence des I2C et SPIDEV.

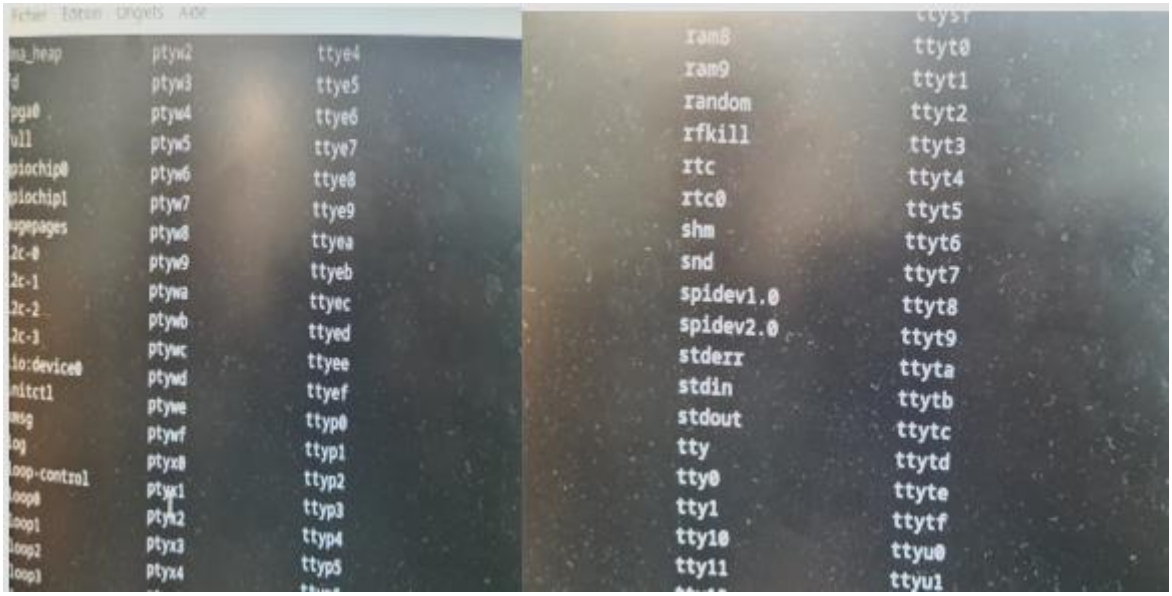


Figure 13 - vérification de l'existence de spi et i2c à l'aide de la commande ls /dev/

Par conséquent, ces résultats indiquent que nous pourrons créer des applications pour travailler avec les deux (i2c et spi).

A ce stade, nous pouvons également utiliser les outils I2C pour détecter ce qui est connecté aux bus I2C et interagir avec eux si nécessaire.

En exécutant la commande **sudo i2cdetect -l**, la liste de tous les contrôleurs I2C enregistrés dans le système est affichée. Pour voir ce qui est connecté à un bus I2C particulier, il faut exécuter la commande **sudo i2cdetect -y -r <0/1/2/3>**, par conséquent le capteur de température connecté à l'I2C-3.

Figure 14 - exécution de la commande sudo i2cdetect -y -r 3

Pour interagir avec le capteur à cette adresse, il suffit de lire le registre "**whoami**", cela peut être réalisé en utilisant la commande **sudo i2cget 3 0x3f 0x01**, cela permet de récupérer du bus I2C-3, à l'adresse du périphérique 0x3f, le contenu du registre 0x01 qui est le registre "**whoami**" du capteur de température, et selon la datasheet, la valeur attendue est 0xa0.

Figure 15 - exécution de la commande sudo i2cget 3 0x3f 0x01

Par conséquent, tous les outils sont disponibles pour créer une application en utilisant i2c et spi. Il suffit d'utiliser un code similaire à celui développer précédemment pour le capteur de température i2c et spi, donc en fessant un build du code dans vitis, un fichier exécutable. elf est générer donc pour l'exécuter sur la carte il faut l'envoyer du PC vers la carte en utilisant l'adresse IP qu'on a configurer avant. Les commandes utilisées sont : **cp nom_fichier.elf ~** pour le copier dans un répertoire où il est facile de le récupérer à l'aide de la carte zuboard, et la commande **scp**

se01@adress_ip_du_pc:~/ nom_fichier.elf . Pour le récupérer sur la carte et par conséquent il suffit de l'exécuter en utilisant la commande **sudo ./nom_fichier.elf** pour que l'accès à i2c et spi soit possible il faut utiliser le sudo.

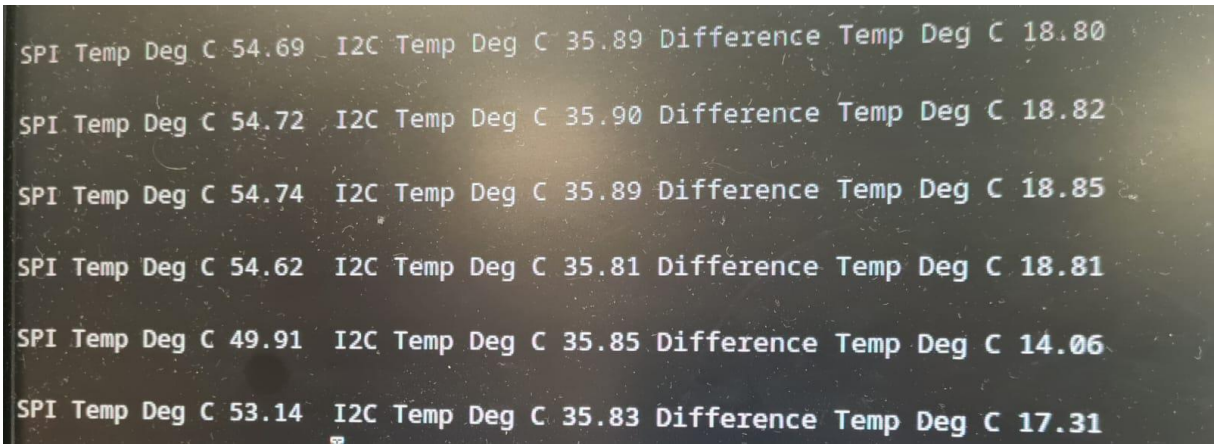


Figure 16 - exécution du code de test de spi et i2c sur la carte.

Le code source pour le test de cette partie se trouve aussi en annexe (3.Code source .c pour le test de température SPI et I2C avec petalinux).

5. Mise en place d'une distribution personnalisée avec Xilinx Open Source Linux

Dans cette quatrième étape de notre projet, nous avons entrepris le développement d'une distribution personnalisée du système d'exploitation en utilisant les fichiers de Xilinx Open Source Linux.

La première étape consiste à récupérer les fichiers sources du noyau Linux depuis le référentiel GitHub de Xilinx, en particulier du répertoire *"linux-xlnx"*. Ce répertoire occupe une position centrale dans le processus de développement, de configuration et de compilation du noyau Linux. On y trouve les sources complètes spécifiquement adaptées aux architectures prises en charge par Xilinx, notamment l'architecture ARM64, pertinente pour notre carte. Le noyau a été obtenu en utilisant la commande ***git clone*** :

```
git clone https://github.com/Xilinx/linux-xlnx.git
```

Pour permettre le développement sur l'architecture ARM64, le cross-compilateur *'aarch64-buildroot-linux-gnu'* a été installé. La compilation du noyau est basée sur un fichier Makefile personnalisé. Des shell scripts tels que ***go*** et ***goinstall*** ont été utilisés pour automatiser le processus. Ces scripts ont été importés depuis un autre TP et ont été modifiés pour être compatibles avec notre architecture. Cela a impliqué la spécification du cross-compilateur en utilisant la variable d'environnement **CROSS_COMPILE=aarch64-buildroot-linux-gnu**, ainsi que la définition de l'architecture **ARCH=arm64**.

D'autre shell scripts utiles ont était utilisés :

- **goconfig** : utilise l'outil de configuration *'menuconfig'* afin de configurer diverses options du noyau, telles que les fonctionnalités du noyau, les pilotes, les paramètres de performance, etc.

```
make ARCH=arm64 CROSS_COMPILE=aarch64-buildroot-linux-gnu- menuconfig
```

- **goulmage** : utilise la commande *mkimage* pour créer une image *uImage* à partir de l'image du noyau Linux compilé afin qu'il soit compatible avec l'u-boot.

```
mkimage -n "Kernel Image' -A arm64 -O linux -C none -T Kernel -a 0x8000 -e 0x8000 -d arch/arm64/boot/Image arch/arm64/boot/uImage
```

Ensuite, on a généré le rootfs à partir du Ramdisk d'un autre TP Codesign en modifiant les shell scripts utilisés (godefconfig, go et goinstall) pour notre architecture. Finalement, on a recopié le device tree depuis le répertoire de Petalinux pour faire fonctionner le module SPI.

Les fichiers nécessaires au démarrage (uImage, uramdisk.image et devicetree_zu.dtb) ont été copiés par le script **goinstall** dans le répertoire *tftpboot*, qui a été utilisé pour le boot de la carte plutôt que l'utilisation de la carte SD. Pour ce faire, des modifications ont été apportées à l'U-boot, notamment la définition des adresses mémoire du kernel, du ramdisk et du fdt, ainsi que des shell scripts *gok*, *gor*, *gotb* qui permettent d'installer ces fichiers aux bons emplacements de la mémoire.

```
setenv kernel_addr_r  0x18000000
setenv ramdisk_addr_r 0x2100000
setenv fdt_addr_r     0x2000000
setenv bootargs console=ttyPS0,115200 root=/dev/ram ramdisk_size=131072
rw earlyprintk earlycon
setenv gok   tftpboot \$\{kernel_addr_r\} uImage
setenv gor   tftpboot \$\{ramdisk_addr_r\} uramdisk.image.gz
setenv godtb tftpboot \$\{fdt_addr_r\} devicetree_zu.dtb
setenv go bootm \${kernel_addr_r} \${ramdisk_addr_r} \${fdt_addr_r}
setenv ramboot run gok \; run gor \; run godtb \; run go
setenv bootcmd run ramboot
saveenv
```

Figure 17 - Configuration d'u-boot

Nous avons ainsi correctement installé le noyau Linux sur la carte avec le script *'run ramboot'*.

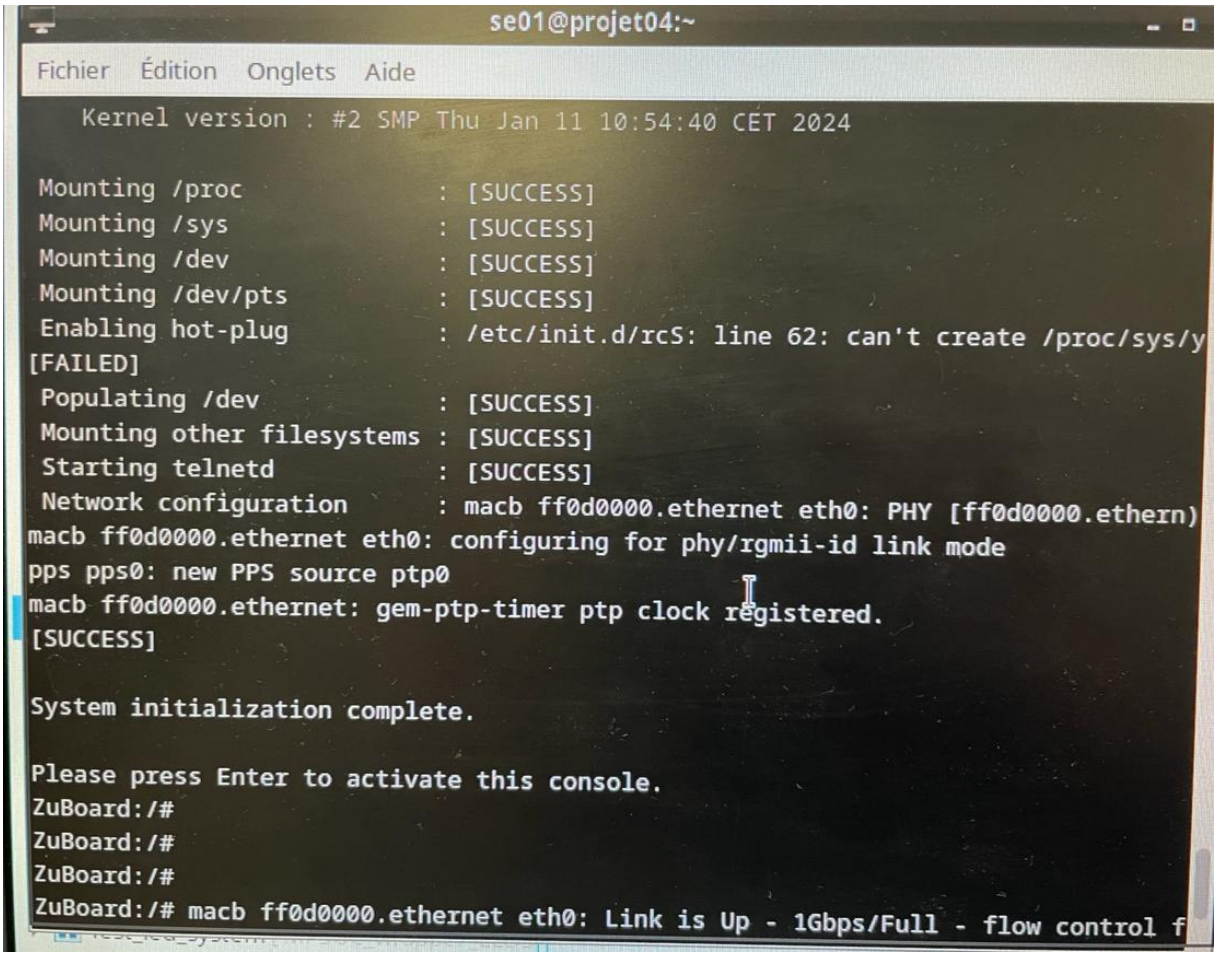


Figure 18 - Distribution Xilinx Linux sur la carte cible

La dernière étape consiste à envoyer des fichiers de tests pour vérifier le fonctionnement du projet. Pour ce faire, nous avons établi la connexion Ethernet avec la carte à travers la commande *'ifconfig'*, puis, à l'aide de la commande *'tftp'*, nous avons transféré les fichiers exécutables *hello_world.elf*, qui affiche un simple 'hello world !', et *test_SPI.elf*, qui a été modifié par rapport à celui développé avec Vitis :

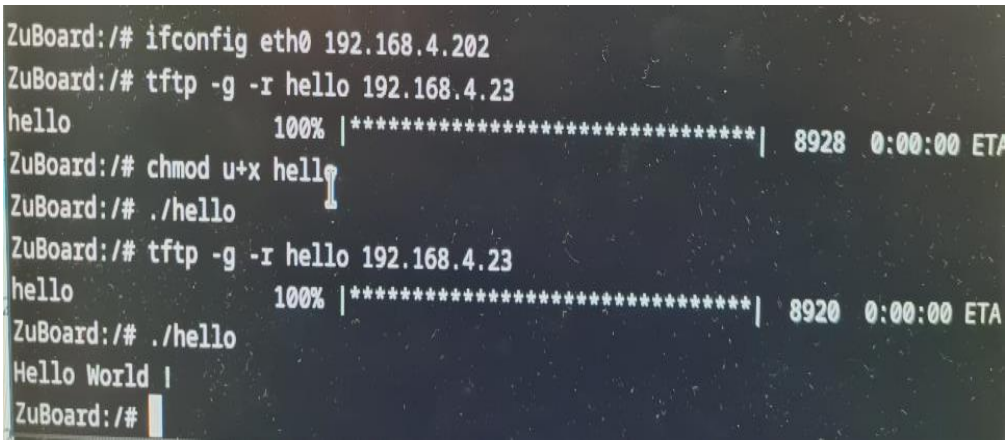


Figure 19 - test hello world avec la distribution personnalisée

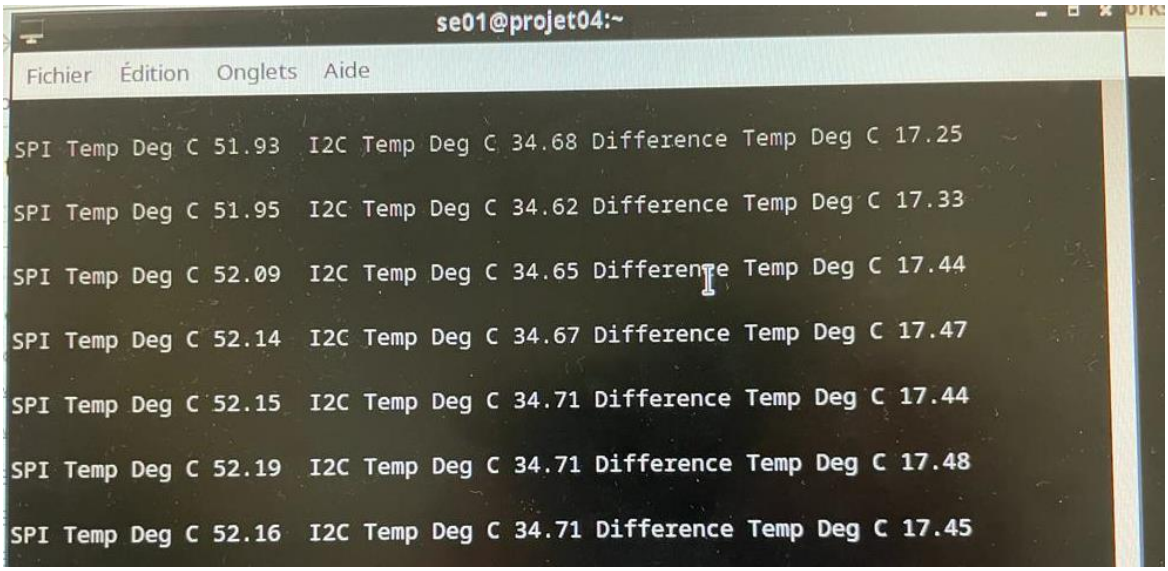


Figure 20 - Test SPI et I2C pour les données de température

6. Conclusion

En conclusion, ce projet de codesign sur la carte de développement Avenet ZuBoard a été une exploration enrichissante de la conception matérielle et logicielle, mettant en lumière les capacités et les caractéristiques de cette plateforme récente. Malgré le manque de documentation abondante spécifique à la ZuBoard, nous avons réussi à tirer parti des similitudes avec la ZedBoard et à adapter les tutoriels existants pour pour tester les périphériques et les capteurs de la carte.

Nous avons réussi à maîtriser l'utilisation des GPIOs, en mettant particulièrement l'accent sur ceux liés à la partie PS (Processing System). La configuration des LEDs, boutons poussoirs et switches a été un point de départ logique pour la prise en main de la carte. Le passage à la communication avec les capteurs, en particulier la lecture de la température par I2C et SPI, a été une étape cruciale où nous avons intégré des blocs IP dans notre design par Vivado.

L'utilisation de Petalinux et de notre propre distribution pour faire fonctionner le module SPI a ajouté une dimension logicielle significative au projet. La configuration du noyau Linux, la gestion des pilotes SPI et I2C, ainsi que le développement d'applications utilisant ces interfaces ont élargi notre compréhension des systèmes embarqués.

7. Annexe

1. Code source .c pour le test GPIO

```
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "xgpiops.h"
#include "xil_io.h"
#include "xparameters.h"
#include "sleep.h"

#define GPIO_BASEADDR XPAR_PSU_GPIO_0_BASEADDR
#define LED1 7
#define LED2 24
#define LED3 25
#define LED4 33

#define PB 32

#define SW1 44
#define SW2 40
#define SW3 39
#define SW4 31

int main()
{
    //Declaration des variables//
    int Status;
    XGpioPs Gpio;
    XGpioPs_Config *Config_Ptr;
    int PB_value;
    int value1;
    int value2;
    int value3;
    int value4;

    //Affectation des variables//


    //Initialisation//
    init_platform();
    Config_Ptr = XGpioPs_LookupConfig(XPAR_XGPIOPS_0_DEVICE_ID);
    Status = XGpioPs_CfgInitialize(&Gpio,Config_Ptr,Config_Ptr->BaseAddr);
    if(Status != XST_SUCCESS){
        return(XST_FAILURE);
    }
```

```
    //LEDs
    XGpioPs_SetDirectionPin(&Gpio, LED1, 1);
    XGpioPs_SetDirectionPin(&Gpio, LED2, 1);
    XGpioPs_SetDirectionPin(&Gpio, LED3, 1);
    XGpioPs_SetDirectionPin(&Gpio, LED4, 1);

    XGpioPs_SetOutputEnablePin(&Gpio, LED1,1);
    XGpioPs_SetOutputEnablePin(&Gpio, LED2,1);
    XGpioPs_SetOutputEnablePin(&Gpio, LED3,1);
    XGpioPs_SetOutputEnablePin(&Gpio, LED4,1);

    //Push Button
    XGpioPs_SetDirectionPin(&Gpio, PB, 0);

    XGpioPs_SetOutputEnablePin(&Gpio, PB,0);

    //Switch
    XGpioPs_SetDirectionPin(&Gpio, SW1, 0);
    XGpioPs_SetDirectionPin(&Gpio, SW2, 0);
    XGpioPs_SetDirectionPin(&Gpio, SW3, 0);
    XGpioPs_SetDirectionPin(&Gpio, SW4, 0);

    XGpioPs_SetOutputEnablePin(&Gpio, SW1,0);
    XGpioPs_SetOutputEnablePin(&Gpio, SW2,0);
    XGpioPs_SetOutputEnablePin(&Gpio, SW3,0);
    XGpioPs_SetOutputEnablePin(&Gpio, SW4,0);

    //Programme//
    print("GPIO\n\r");
    while(1){
        PB_value = XGpioPs_ReadPin(&Gpio, PB);
        value1 = XGpioPs_ReadPin(&Gpio, SW1);
        value2 = XGpioPs_ReadPin(&Gpio, SW2);
        value3 = XGpioPs_ReadPin(&Gpio, SW3);
        value4 = XGpioPs_ReadPin(&Gpio, SW4);
        //xil_printf("%d\n",PB_value);
        XGpioPs_WritePin(&Gpio, LED1, value1 && PB_value);
        XGpioPs_WritePin(&Gpio, LED2, value2 && PB_value);
        XGpioPs_WritePin(&Gpio, LED3, value3 && PB_value);
        XGpioPs_WritePin(&Gpio, LED4, value4 && PB_value);
        usleep(100000);
    }
    cleanup_platform();
    return 0;
}
```

2. Code source .c pour le test de température I2C

```
#include <stdio.h>

#include "platform.h"
#include "xil_printf.h"
#include "xiic.h"

#define IIC_dev             XPAR_IIC_0_DEVICE_ID
#define IIC_SLAVE_ADDR      0x3F
#define BUFFER_SIZE         6
#define WHOAMI              0xA0

XIic  iic;
u8 SendBuffer [2];
u8 RecvBuffer [2];
u16 result;
float temp;

int main()
{
    XIic_Config *iic_conf;

    init_platform();

    print("Hello World\n\r");
    print("Successfully ran Hello World application");


    iic_conf = XIic_LookupConfig(IIC_dev);
    XIic_CfgInitialize(&iic, iic_conf, iic_conf->BaseAddress);

    SendBuffer[0] = 0x01;
    XIic_Send(iic.BaseAddress,IIC_SLAVE_ADDR,(u8 *)&SendBuffer,
1,XIIC_REPEATED_START);
    XIic_Recv(iic.BaseAddress,IIC_SLAVE_ADDR,(u8 *)&RecvBuffer, 1,XIIC_STOP);

    if(RecvBuffer[0]==WHOAMI){
        printf("Temp Sensor Detected\n\r");
    }
    else{
        printf("Temp Sensor NOT Detected\n\r");
        return 0;
    }

    SendBuffer[0] = 0x04;
    SendBuffer[1] = 0x0C;
    XIic_Send(iic.BaseAddress,IIC_SLAVE_ADDR,(u8 *)&SendBuffer, 2,XIIC_STOP);

    SendBuffer[0] = 0x04;
```

```
    XIic_Send(iic.BaseAddress,IIC_SLAVE_ADDR,(u8 *)&SendBuffer,
1,XIIC_REPEATED_START);
    XIic_Recv(iic.BaseAddress,IIC_SLAVE_ADDR,(u8 *)&RecvBuffer, 1,XIIC_STOP);

    while (1){
        SendBuffer[0] = 0x06;
        XIic_Send(iic.BaseAddress,IIC_SLAVE_ADDR,(u8 *)&SendBuffer,
1,XIIC_REPEATED_START);
        XIic_Recv(iic.BaseAddress,IIC_SLAVE_ADDR,(u8 *)&RecvBuffer,
sizeof(RecvBuffer),XIIC_STOP);
        result = RecvBuffer[1] << 8 | RecvBuffer[0];
        temp = (float) result / 100;
        printf("Temperature is %f \n\r",temp);
        usleep(1000000);

    }


    cleanup_platform();
    return 0;
}
```

3. Code source .c pour le test de température SPI et I2C avec petalinux

```c
#include <stdint.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <getopt.h>
#include <fcntl.h>
#include <time.h>
#include <sys/ioctl.h>
#include <linux/ioctl.h>
#include <sys/stat.h>
#include <linux/types.h>
#include <linux/spi/spidev.h>
#include <linux/i2c-dev.h>
#include <linux/i2c.h>

#define ARRAY_SIZE(array) sizeof(array)/sizeof(array[0])

static const char *spi_device = "/dev/spidev1.0";
static const char *i2c_device = "/dev/i2c-3";
static const char address = 0x3F;

static uint8_t bits = 8;
static uint32_t speed = 1000000;
static uint32_t mode =SPI_MODE_3 ;
static uint16_t delay;

int main(){
    int spi;
    int i2c;
    int ret = 0;
    unsigned char spi_tx_buf[2];
    unsigned char spi_rx_buf[2];
    unsigned char i2c_tx_buf[2];
    unsigned char i2c_rx_buf[2];

    int temp;
    float spi_temp_degC;
    int16_t spi_temperature;
    float i2c_temp_degC;
    int16_t i2c_temperature;
    char temp_l;
    char temp_h;


    struct i2c_msg msgs[2];
```

```
    struct i2c_rdwr_ioctl_data msgset[1];

    struct spi_ioc_transfer tr = {
            .tx_buf = (unsigned long)spi_tx_buf,
            .rx_buf = (unsigned long)spi_rx_buf,
            .len = ARRAY_SIZE(spi_tx_buf),
            .delay_usecs = delay,
            .speed_hz = 0,
            .bits_per_word = 0,
        };

//Open SPI and I2C
    spi = open(spi_device, O_RDWR);
    if (spi < 0)
        printf("can't open SPI device\n\r");

    i2c = open(i2c_device, O_RDWR);
    if (i2c < 0)
        printf("can't open I2C device\n\r");

//configure SPI
    ret = ioctl(spi, SPI_IOC_WR_MODE32, &mode);
    if (ret == -1)
        printf("can't set spi mode\n\r");

    ret = ioctl(spi, SPI_IOC_RD_MODE32, &mode);
    if (ret == -1)
        printf("can't get spi mode\n\r");

    ret = ioctl(spi, SPI_IOC_WR_BITS_PER_WORD, &bits);
    if (ret == -1)
        printf("can't set bits per word\n\r");

    ret = ioctl(spi, SPI_IOC_RD_BITS_PER_WORD, &bits);
    if (ret == -1)
        printf("can't get bits per word\n\r");

    ret = ioctl(spi, SPI_IOC_WR_MAX_SPEED_HZ, &speed);
    if (ret == -1)
        printf("can't set max speed hz\n\r");

    ret = ioctl(spi, SPI_IOC_RD_MAX_SPEED_HZ, &speed);
    if (ret == -1)
        printf("can't get max speed hz\n\r");

//enable SPI sensor sampling
    spi_tx_buf[0] = (char) 0x10; //enable sampling
    spi_tx_buf[1] = (char) 0x20;
```

```
    ret = ioctl(spi, SPI_IOC_MESSAGE(1), &tr);
    if (ret == -1)
            printf("IOCTL error\n\r");

//check SPI temp sensor can be detected
    spi_tx_buf[0] = (char) 0x8F; // read who am I
    spi_tx_buf[1] = (char) 0x00;

    ret = ioctl(spi, SPI_IOC_MESSAGE(1), &tr);
    if (ret == -1)
            printf("IOCTL error\n\r");
    if (spi_rx_buf[1] == 0xb3)
        printf("LPS22 Detected\n\r");

//check I2C temp sensor can be detected
    i2c_tx_buf[0] = 0x01;

    msgs[0].addr = address;
    msgs[0].flags = 0;
    msgs[0].len = 1;
    msgs[0].buf = i2c_tx_buf;

    msgs[1].addr = address;
    msgs[1].flags = I2C_M_RD;
    msgs[1].len = 1;
    msgs[1].buf = i2c_rx_buf;

    msgset[0].msgs = msgs;
    msgset[0].nmsgs = 2;

    if (ioctl(i2c, I2C_RDWR, &msgset) < 0) {
        perror("ioctl(I2C_RDWR) in i2c_read");
    }

    if (i2c_rx_buf[0] == 0xa0)
        printf("STTS22H Detected\n\r");


//set sampling on the I2C temp sensor
    i2c_tx_buf[0] = 0x04;
    i2c_tx_buf[1] = 0x0c;

    msgs[0].addr = address;
    msgs[0].flags = 0;
    msgs[0].len = 2;
    msgs[0].buf = i2c_tx_buf;

    msgset[0].msgs = msgs;
    msgset[0].nmsgs = 1;
```

```
    if (ioctl(i2c, I2C_RDWR, &msgset) < 0)
        perror("ioctl(I2C_RDWR) in i2c_write");

while(1){

//read temperature from the SPI sensor
        spi_tx_buf[0] = (char) 0xab;
        spi_tx_buf[1] = (char) 0x00;

        ret = ioctl(spi, SPI_IOC_MESSAGE(1), &tr);
        if (ret == -1)
                printf("IOCTL error\n\r");

        temp_l = spi_rx_buf[1];


        spi_tx_buf[0] = (char) 0xac;
        spi_tx_buf[1] = (char) 0x00;

        ret = ioctl(spi, SPI_IOC_MESSAGE(1), &tr);
        if (ret == -1)
                printf("IOCTL error\n\r");
        temp_h = spi_rx_buf[1];

        temp = ((temp_h <<8)|(temp_l));

        if ((temp & 0x8000) == 0) //msb = 0 so not negative
            {
                spi_temperature = temp;
            }
            else
            {
                // Otherwise perform the 2's complement math on the value
                spi_temperature = (~(temp - 0x01)) * -1;
            }
        spi_temp_degC = (float) spi_temperature /100.0;

//Read temperature from the I2C sensor
        i2c_tx_buf[0] = 0x06;

        msgs[0].addr = address;
        msgs[0].flags = 0;
        msgs[0].len = 1;
        msgs[0].buf = i2c_tx_buf;

        msgs[1].addr = address;
        msgs[1].flags = I2C_M_RD;
        msgs[1].len = 2;
```

```
        msgs[1].buf = i2c_rx_buf;

        msgset[0].msgs = msgs;
        msgset[0].nmsgs = 2;

        if (ioctl(i2c, I2C_RDWR, &msgset) < 0) {
            perror("ioctl(I2C_RDWR) in i2c_read");
        }

        i2c_temperature = i2c_rx_buf[1] << 8 | i2c_rx_buf[0];
        i2c_temp_degC = (float) i2c_temperature / 100;

        printf("SPI Temp Deg C %4.2f  I2C Temp Deg C %4.2f Difference Temp Deg
C %4.2f  \n\n\r",spi_temp_degC, i2c_temp_degC, (spi_temp_degC - i2c_temp_degC)
);



        usleep(1000000);
    }
}
```

4. Références

Pour la carte Avnet ZuBoard

https://www.avnet.com/wps/portal/us/products/avnet-boards/avnet-board-families/zuboard-1cg/

Pour la datasheet du capteur de température

https://www.alldatasheet.com/datasheet-pdf/pdf/1284633/STMICROELECTRONICS/STTS22HTR.html

Pour petalinux

https://www.adiuvoengineering.com/post/microzed-chronicles-ioctl-spi-i2c-and-petalinux

https://docs.xilinx.com/r/en-US/ug1144-petalinux-tools-reference-guide/Configuring-Device-Tree

Pour Xilinx Open Source Linux

https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/460653138/Xilinx+Open+Source+Linux

https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842481/Build+kernel