[FPGA]
Linux enabled, HW accelerator

PROFESSEUR RÉFÉRENT :

Dr THIÉBOLT François

Ingénieur de Recherche, UT3
Centre de Ressources Informatiques
Architecture Systèmes Réseaux (ASR) - Équipe SEPIA
Université Toulouse III - Paul Sabatier
05 61 55 88 67
Francois.Thiebolt@irit.fr

AUTEURS :

HERZBERG Dwayne

Master parcours Systèmes embarqués et connectés
infrastructures et logiciels (SECIL),
Université Toulouse III - Paul Sabatier

PAILLOT Oleg

Master parcours Systèmes embarqués et connectés
infrastructures et logiciels (SECIL),
Université Toulouse III - Paul Sabatier

20/08/2024

Résumé

Ce rapport présente un projet de TER (Travail d'Étude et de Recherche) intitulé "Linux Enabled HW Accelerator sur FPGA", réalisé dans le cadre du Master SECIL à l'Université Paul Sabatier - Toulouse III. L'objectif principal de ce projet est de concevoir et d'implémenter un environnement de travaux pratiques prêt à l'emploi pour les étudiants de master, en utilisant des systèmes FPGA reconfigurables et un système Linux embarqué. Le projet vise à exploiter les capacités des FPGA, en particulier les cartes Xilinx Zybo Z7-20, pour permettre aux étudiants d'apprendre et de comprendre la description matérielle et son interaction avec Linux.

Les étapes clés incluent la configuration matérielle via Xilinx Vivado, la création d'un environnement Linux avec l'outil PetaLinux, la configuration d'un système de fichiers racine (RFS), la création de blocs IP en VHDL, ainsi que le développement d'applications sous Linux. Ce rapport détaille le processus de boot, l'utilisation de l'arborescence des périphériques, et la gestion des FPGA, illustrée par l'implémentation du module PmodENC. Une attention particulière est accordée à l'accélération matérielle : le FPGA gère l'incrémentation et la décrémentation d'un compteur en réponse aux signaux de l'encodeur rotatif, tandis que le processeur exécutant Linux se charge uniquement de la lecture de ce compteur. Cette division des tâches montre l'efficacité et les avantages de l'accélération matérielle par rapport à une implémentation purement logicielle.

En outre, le rapport présente en détail l'implémentation d'un système Linux embarqué, ainsi que le co-design entre le FPGA et Linux. Pour assurer une compréhension et une reproductibilité complète, l'ensemble des travaux, procédures, et concepts ont été expliqués et documentés.

Table des matières

1 Introduction

A l'occasion des TER du Master Informatique, nous avons pu réaliser un projet sur les FPGA et Linux embarqué sous la tutelle de M. THIEBOLT, professeur à l'Université Toulouse III. La durée de ce projet est fixée à 3 mois et, hormis les différents livrables techniques, un rapport et une soutenance doivent être réalisés. Ce document fait donc suite à cette première demande et détaille l'avancement accompli durant ces quelques mois de travail.

1.1 Contexte

Les FPGA font partie intégrante du parcours SECIL du Master Informatique proposé à l'UT3. Au moment où ce document est écrit, des cartes Zybo Z7-20 sont utilisées comme outil pédagogique afin d'apprendre les bases de la chaîne de développement Xilinx et, de façon plus générale, celle des FPGA.

Par ailleurs, il se trouve que les circuits Zynq présents sur ces cartes sont des SoC. Il est donc ainsi possible d'implanter un système Linux sur ces cartes, ouvrant la porte à une reprogrammation dynamique de la partie PL en passant par Linux.

C'est alors qu'une idée est apparue, celle de mettre en place des environnements de TP orientés FPGA reconfigurables en passant par Linux, et ce à distance. Mais avant d'en arriver là, il serait intéressant d'expérimenter d'abord la mise en place d'une simple accélération matérielle, c'est-à-dire une interaction entre la partie PL et PS, en passant par une reprogrammation dynamique du PL sous Linux. Cela donnerait lieu à une documentation permettant de plus facilement mettre en place cette idée initiale.

Par conséquent, la création d'un co-design PL/PS orienté sur l'utilisation d'un encodeur rotatif disponible sur les Zybo, le PmodENC, a été choisi. Il nous revient donc de mener à terme ce projet d'accélération matérielle tout en gardant une documentation à jour.

Une tentative de mettre un système Linux personnalisé sur la Zybo Z7-20 a déjà été réalisée par un ancien groupe en 2018, toutefois la documentation a largement été perdue. D'autre part, la partie reprogrammation dynamique de la partie PL n'a visiblement pas été abordée.

1.2 Objectifs

L'objectif final est d'avoir une accélération matérielle du PmodENC grâce au co-design PL/PS et ce en reprogrammant dynamiquement la partie PL depuis Linux, cette dernière n'a donc pas son bitstream injecté dès le lancement de la Zybo.

Nous avons avancé petit à petit, en décomposant au fur et à mesure ce but final en plusieurs étapes afin d'arriver à un ensemble fonctionnel. Lesdites étapes sont visibles dans notre plan, annoncé ci-dessous.

1.3 Plan

Nous aborderons d'abord l'implémentation du RFS Linux personnalisé, des explications seront données quant au fonctionnement du boot sur les Zybo et sur les différents éléments qui constituent ce procédé.

Ensuite, le fonctionnement du PmodENC et son implémentation sur la Zybo sera étudié, différentes approches ont été utilisées afin de mieux comprendre l'outil et d'arriver à l'approche qui nous permettra d'arriver à nos fins.

Enfin, le sujet de la reprogrammation dynamique sera évoqué et sera le dernier élément nécessaire au bon déroulement de l'accélération matérielle. Là aussi, nos premières démarches seront mentionnées, même si seulement une seule sera retenue pour ce projet.

2 Gestion de projet

2.1 Organisation

L'organisation du travail au sein de notre projet a reposé sur une approche flexible et structurée, s'adaptant aux spécificités d'une équipe réduite et aux exigences exploratoire du projet. Les séances de travail se sont principalement déroulées en présentiel dans la salle U3-305, un environnement qui a favorisé les échanges directs et immédiats. L'emploi du temps adopté s'étendait du lundi au vendredi, de 10h à 17h. Ce planning régulier a servi de cadre de référence tout au long du projet. Nous avons néanmoins conservé une certaine flexibilité dans cet emploi du temps, permettant de moduler les séances de travail en fonction des contraintes personnelles et des besoins ponctuels du projet.

En raison de la taille réduite de l'équipe, composée de seulement deux membres, il n'y avait pas de répartition rigide des rôles. Chacun des membres a endossé plusieurs responsabilités, agissant comme développeur, documentariste, ou gestionnaire de projet. Permettant ainsi une couverture complète du projet. La répartition des tâches joint à une communication et une journalisation du travail constante nous a permis de rester synchronisés sur les objectifs et les progrès tout en travaillant de manière autonome lorsque cela était nécessaire.

Les objectifs initiaux du projet ont progressivement évolué pour se concrétiser en exigences à remplir. Ces exigences ont servi de fondement à l'organisation et à la planification du travail. Nous avons mis en place une approche itérative, articulée autour de cycles successifs de recherche, de développement, de tests, et d'analyse. La figure ci-dessous illustre ce processus, montrant comment nous avons commencé par la recherche et la documentation, suivi de la mise en place de l'environnement de développement. Ensuite, nous avons effectué des exécutions et des tentatives, intégrant des phases de feedback et de logging pour affiner notre travail, jusqu'à atteindre la fin du projet.

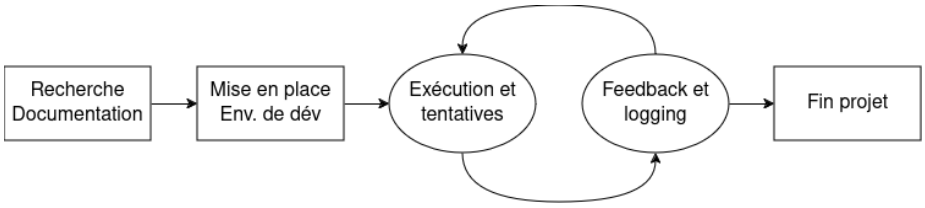

FIGURE 1 – Processus itératif

2.2 Environnement de travail

Le tableau 1 présente une liste exhaustive des outils utilisés dans le cadre de ce projet.

L'ensemble des projets développés avec Vivado a été initialement réalisé sous la version 2023.1, installée sur les postes de travail de l'université. Par la suite, ces projets ont été portés avec succès vers la version 2023.2, la dernière disponible lors de la réalisation de ce TER.

Tout le processus de travail, de la compréhension à l'implémentation, ainsi que l'ensemble des technologies, méthodes, protocoles, ont été documentés et consignés sur une documentation [2] en ligne dédiée. Elle offre non seulement une vue d'ensemble du projet, mais détaille également chaque outil utilisé et chaque étape suivie permettant une reproductibilité du projet.

En complément, un journal de bord a été maintenu tout au long du projet, permettant de suivre de manière régulière l'avancement des travaux.

Les travaux réalisés ont été consigné et expliqué dans plusieurs répertoires GitHub au sein de l'organisation dédiée à ce TER [3]. Cette organisation regroupe l'ensemble du code source, des scripts, et explications nécessaires à la reproduction et à l'amélioration des projets.

Vivado 2023.1, Vivado 2023.2	Git
Vitis	Visual Studio et ses extensions
XSCT	mkdocs
Bootgen	mkdocs-material
PetaLinux	
Device Tree Generator	
GCC Cross-compiler	
RFS Debian (EEWiki)	
fdisk	
mkfs	
Librairie Digilent pour Vivado	
Zybo Z7-20	
Digilent PmodENC	

TABLE 1 – Outils utilisés dans le projet

3 Linux

L'un des principaux objectifs de ce TER est l'installation d'un RFS Linux personnalisé, typiquement l'une des grandes distributions connues telles que Debian, Ubuntu ou Fedora, sur la Zybo Z7-20 et de créer une documentation complète sur le procédé. En effet, l'installation diffère de celle classiquement pratiquée sur les systèmes embarqués de type SBC de par l'interaction entre le processeur Cortex-A9 et le FPGA Zynq-7000. Des explications seront données sur le fonctionnement d'un système de boot initialement afin de mieux appréhender les différents éléments de notre implémentation.

3.1 Séquence de boot

Le processus de boot d'un système embarqué sur la Zybo Z7-20 est organisé en plusieurs étapes. Cette séquence, qui passe par des phases successives d'initialisation du matériel et de configuration logicielle, garantit un démarrage ordonné et fonctionnel du système. Chaque étape dépend des précédentes pour assurer que tous les composants nécessaires sont prêts à fonctionner.

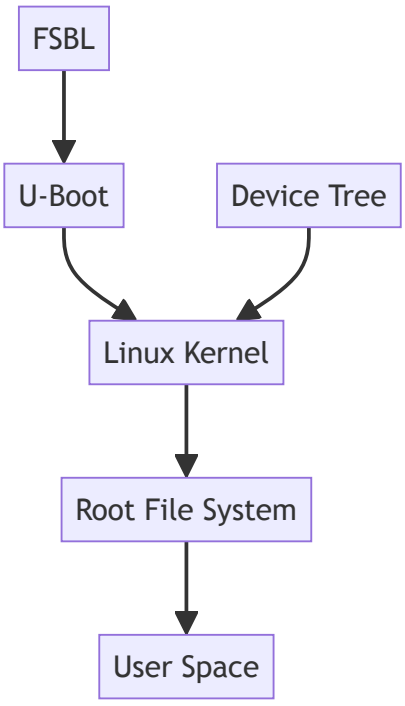

FIGURE 2 – Séquence de boot implémentée

3.1.1 First Stage Bootloader (FSBL)

Le processus de boot commence avec le FSBL (First Stage Boot Loader). En tant que premier programme exécuté après la mise sous tension du système, le FSBL est chargé depuis une mémoire non volatile, comme la QSPI flash. Il est important de noter que dans le cas de la carte Zybo, le FSBL est en fait chargé depuis la carte SD, comme mentionné au chapitre 2 du manuel Zybo. Le rôle du FSBL est crucial : il initialise les composants matériels de base, tels que la mémoire DDR et les périphériques critiques, qui sont indispensables pour les étapes suivantes.

Une fois cette initialisation matérielle de base complétée, le FSBL transfère le contrôle au second stage bootloader, généralement U-Boot. Cette transition est nécessaire car U-Boot requiert un environnement matériel déjà préparé pour fonctionner. Cependant, il est intéressant de noter que U-Boot possède un outil, U-Boot SPL (Secondary Program Loader), qui peut également jouer le rôle de FSBL. Dans notre cas spécifique, nous utilisons le FSBL fourni par Xilinx plutôt que le U-Boot SPL pour réaliser cette première étape d'initialisation.

3.1.2 Universal Bootloader (U-Boot)

U-Boot (Das U-Boot) est bootloader open-source largement utilisé dans les systèmes embarqués. Il prend le relais après le FSBL et assume des responsabilités plus complexes. Il continue le processus d'initialisation matérielle en configurant les périphériques supplémentaires qui n'ont pas été couverts par le FSBL. De plus, U-Boot charge en mémoire le kernel Linux, le DTB et le RFS, éléments nécessaires pour démarrer le système d'exploitation.

La capacité d'U-Boot à gérer ces tâches permet une transition vers le kernel Linux. Il garantit que tous les composants matériels et configurations nécessaires sont en place pour que le kernel prenne le relais.

3.1.3 Linux Kernel

Le kernel Linux peut maintenant être chargé en mémoire et démarrer. Ce noyau, cœur du système d'exploitation, prend en charge la gestion des ressources matérielles et l'exécution des processus. À ce stade, il s'appuie sur les informations du Device Tree pour identifier et initialiser correctement les périphériques matériels présents sur la carte.

En plus du kernel, un autre élément crucial lors du démarrage est l'initramfs (Initial RAM Filesystem). L'initramfs est une image de système de fichiers temporaire qui est chargée en mémoire et utilisée par le kernel lors des premières étapes de son exécution. Il contient les scripts d'initialisation et les pilotes nécessaires pour permettre au système d'accéder au véritable RFS.

Une fois que l'initramfs a terminé son rôle, le kernel monte le RFS, qui contient tous les fichiers nécessaires au fonctionnement du système. Cette étape

prépare le lancement du processus initial, souvent appelé "init", qui démarre tous les autres processus utilisateurs et services du système.

3.1.4 Device Tree

Le Device Tree décrit le matériel présent sur la carte et fournit au kernel Linux les informations nécessaires pour configurer et gérer ces périphériques. En maintenant le kernel générique, le Device Tree permet d'utiliser le même kernel sur différentes plateformes matérielles, simplement en modifiant le fichier de description matérielle.

3.1.5 Root File System (RFS)

Après l'initialisation du matériel par le kernel linux, le RFS est monté. Ce système de fichiers contient le contexte et les ressources nécessaires pour que le kernel Linux puisse exécuter des applications et des services. Il inclut des bibliothèques partagées, des utilitaires système, des fichiers de configuration, et diverses applications.

Le montage du RFS est une étape clé qui prépare l'environnement utilisateur, reliant le kernel aux applications de niveau supérieur et aux services qui s'exécuteront en User Space.

3.1.6 User Space

Enfin, avec le système de fichiers racine monté et le processus init lancé, le contrôle passe à l'User Space. Cet espace est l'environnement où toutes les applications et services utilisateurs fonctionnent, indépendamment du kernel. L'User Space inclut toutes les bibliothèques, applications et services nécessaires pour que le système soit pleinement opérationnel et interactif pour les utilisateurs.

Cette dernière étape complète la séquence de boot, assurant que le système est prêt pour son utilisation prévue, avec toutes les ressources matérielles et logicielles correctement initialisées et disponibles.

3.2 Implémentation sur Zybo Z7-20

La Zybo Z7-20 est une carte de développement équipée d'un FPGA Zynq-7000 de Xilinx, combinant un processeur ARM Cortex-A9 bicœur et des ressources logiques programmables. Pour installer Linux sur cette carte, il est crucial de bien comprendre les outils et processus impliqués. L'objectif est de créer un environnement Linux embarqué qui fonctionne avec les capacités matérielles de la carte. Pour ce faire, il est nécessaire d'utiliser Vivado pour générer la spécification matérielle et PetaLinux pour configurer et déployer le système d'exploitation. Ces choix d'outils ne sont pas arbitraires. Ils répondent à des exigences et facilitent la mise en œuvre du système. La procédure et l'ensemble des commandes sont détaillés et accessibles dans la documentation en ligne.

3.2.1 XSA

Le fichier XSA, ou Xilinx Support Archive, est un élément central dans le développement embarqué sur la Zybo Z7-20. Ce fichier, généré par Vivado, contient une description complète de la configuration matérielle, incluant les blocs IP, les contraintes de timing et le périphériques. Ce fichier est essentiel car il sert de pont entre le matériel et le logiciel.

Dans le développement pour des plateformes comme le Zynq, où le FPGA et le processeur ARM sont étroitement intégrés, il est crucial de s'assurer que le logiciel comprend la configuration matérielle pour interagir correctement avec tous les composants. Le XSA garantit cette cohérence en encapsulant toutes les informations nécessaires pour que le système d'exploitation puisse utiliser et interagir efficacement avec le matériel.

Pour réaliser la descritpion matériel les liens entre le processeur Zynq et le FPGA de ce projet, nous avons utilisé le dépôt officiel de la Zybo Z7-20 disponible sur GitHub, en suivant le guide "Digilent FPGA Demo Git Repositories". Ce dépôt contient une démonstration complète du matériel de la Zybo Z7-20, y compris un design matériel prêt à l'emploi. En nous basant sur cet exemple, nous avons pu générer un fichier XSA qui intègre la configuration nécessaire pour notre projet d'implémentation Linux. L'utilisation de ce dépôt a permis de s'assurer que notre configuration matérielle était correcte et optimisée pour la communication entre le processeur et le FPGA et avec l'ensemble des périphériques disponible sur la carte.

3.2.2 Petalinux

Nous souhaitons maintenant monter un système d'exploitation Linux compatible à notre configuration décrite par le fichier XSA précédemment généré.

Nous avons tout d'abord exploré plusieurs outils open-source pour créer un système d'exploitation Linux. Cependant, nous avons rencontré des difficultés à faire fonctionner ces outils de manière fiable sur la Zybo Z7-20. Les solutions comme Yocto ou Buildroot, bien qu'offrant une grande flexibilité, nécessitent des configurations manuelles détaillées pour chaque plateforme, ce qui a ajouté de la complexité et du temps à notre projet.

C'est alors que nous avons découvert PetaLinux, un SDK développé par Xilinx et spécialement dédié à leurs plateformes matérielles. Cette découverte a été proposée à notre professeur, qui a accepté cette nouvelle approche. PetaLinux est conçu pour simplifier le développement de SoC basés sur FPGA en intégrant étroitement le flux de travail de Vivado. Cela nous permet de transférer directement les configurations matérielles définies dans le fichier XSA vers la configuration logicielle. Cette intégration a grandement simplifié le processus de création et de personnalisation d'un système d'exploitation Linux embarqué, tout en garantissant une compatibilité avec les spécificités matérielles des dispositifs Xilinx.

En utilisant PetaLinux, nous avons bénéficié d'une solution qui automatise la création de nombreux composants nécessaires, tels que le FSBL, U-Boot, le kernel Linux et le Device Tree. Cette automatisation a réduit les risques d'erreurs et a assuré une configuration correcte et cohérente du système, adaptée aux exigences spécifiques de notre projet.

3.2.3 RFS

Bien que PetaLinux offre la possibilité de générer un RFS, nous avons décidé de créer un RFS personnalisé pour mieux répondre aux besoins spécifiques de notre projet sur la Zybo Z7-20. Ce choix nous permet de contrôler plus finement les composants inclus dans le système et d'ajouter certains pilotes spécifiques nécessaires à notre future application.

Dans notre démarche pour concevoir un RFS adapté, nous avons exploré plusieurs options. La première option envisagée a été l'utilisation d'ARMbian, une distribution Linux optimisée pour les systèmes embarqués basée sur Debian. Cette suggestion, initialement proposée par notre enseignant, semblait prometteuse en raison de la réputation d'ARMbian pour le support des SBC. Cependant, ARMbian est principalement conçu pour une liste spécifique de cartes, et la Zybo Z7-20 n'en fait pas partie. Adapter ARMbian pour fonctionner avec la Zybo aurait nécessité de modifier significativement le processus de construction du RFS pour qu'il soit compatible avec le matériel et les particularités du projet. Cette approche s'est révélée non-idéale pour notre projet en raison du temps et des ressources supplémentaires qu'elle aurait nécessité.

Nous avons alors exploré une deuxième option : utiliser Debian, une distribution Linux bien établie et flexible, mais dans une version adaptée aux systèmes embarqués. Notre objectif était de disposer d'un RFS Debian minimal, n'incluant que les composants essentiels à notre application, tels que les bibliothèques de base et les utilitaires système, sans environnement de bureau, gestionnaires de fenêtres ou autres logiciels superflus qui sont généralement inutiles dans un environnement embarqué.

Une méthode possible pour réaliser cela est d'utiliser debootstrap, un outil qui permet de construire un système Debian minimal directement depuis les archives Debian. Bien que cette méthode soit puissante et flexible, elle nécessitait des ajustements et des tests pour s'assurer qu'elle fonctionnerait correctement sur notre matériel. Après avoir considéré cette option, nous avons finalement décidé d'utiliser un RFS Debian personnalisé déjà disponible sur le site DigiKey (anciennement eeWiki) [1]. Ce système de fichiers, déjà configuré pour les systèmes ARM, répondait parfaitement à nos besoins. En choisissant cette solution, nous avons pu économiser du temps de développement tout en garantissant que notre RFS était bien adapté aux spécificités des systèmes embarqués. De plus, cela nous a permis d'ajouter facilement les pilotes et modules supplémentaires requis par notre projet.

4 PmodENC

Le but de cette section est de détailler notre démarche pour faire fonctionner le composant PmodENC de Digilent sur la carte Zybo Z7-20. Le PmodENC est un encodeur rotatif, un dispositif utilisé pour capturer les mouvements de rotation et les traduire en signaux numériques que l'on peut ensuite traiter.

Pour exploiter pleinement les capacités de ce composant dans notre projet, nous avons décidé d'explorer différentes possibilités d'implémentation. Notre objectif final est de combiner le design FPGA et le processeur ARM pour créer une application C sous Linux qui utilise un registre géré par le FPGA, permettant ainsi de créer une accélération matérielle.

4.1 Présentation

Le PmodENC de Digilent est un encodeur rotatif avec un bouton poussoir intégré, souvent utilisé dans des projets embarqués pour contrôler des compteurs ou naviguer dans des menus. Sur la carte Zybo Z7-20, il se connecte via le connecteur Pmod et permet de capturer les rotations de l'axe pour générer des signaux que le FPGA peut traiter. Le projet que nous avons réalisé vise à exploiter ce composant pour contrôler un compteur dont la valeur est affichée sur les LEDs de la carte FPGA. Ce projet nous a permis d'explorer différentes approches d'implémentation, de la plus simple, utilisant uniquement la PL, à des solutions plus complexes intégrant des éléments de co-design matériel et logiciel.

4.2 Implémentation

Pour atteindre notre objectif final de co-design matériel et logiciel, nous avons étudié et mis en œuvre plusieurs méthodes d'implémentation du PmodENC sur la Zybo Z7-20. Cela nous a permis d'explorer les différentes possibilités d'implémentation que nous connaissions, afin de les comparer et d'approfondir notre compréhension de la Zybo Z7-20. Les trois principales approches que nous avons explorées sont l'implémentation FPGA pure (PL-only), une implémentation en baremetal, et finalement une implémentation intégrant un bloc IP avec interface AXI.

4.2.1 PL-only

Dans cette approche, nous avons choisi de mettre en œuvre l'ensemble du système en utilisant exclusivement la PL de la carte FPGA. L'objectif était de développer un système capable de gérer les signaux de l'encodeur rotatif PmodENC et de piloter le compteur affiché sur les LEDs, sans recourir au processeur. Cette approche nous a permis de mieux comprendre le fonctionnement de l'encodeur rotatif, en analysant et en traitant directement les signaux qu'il génère. En travaillant de cette manière, nous avons pu commencer à préparer un bloc

IP pour les étapes suivantes du projet, facilitant ainsi l'intégration future de cet encodeur dans un environnement de co-design matériel et logiciel.

Pour ce faire, nous avons conçu plusieurs modules VHDL, chacun ayant un rôle dans le fonctionnement global du système. Le module principal, décrit dans le fichier `pmodenc.vhd`, établit l'interface avec l'encodeur rotatif et coordonne les opérations des autres modules. Le fichier `encoder.vhd` gère la logique de lecture des signaux générés par l'encodeur, en s'assurant de détecter correctement les impulsions et les changements de direction. Enfin, un module de stabilisation des signaux, décrit dans le fichier `debouncer.vhd`, est utilisé pour filtrer les parasites et garantir que les signaux d'entrée sont stables avant d'être traités.

Pour connecter l'encodeur rotatif PmodENC à la carte FPGA, nous avons assigné les pins de l'encodeur aux entrées correspondantes dans notre code VHDL en utilisant un fichier de contrainte XDC. Ce fichier fait correspondre les signaux logiques définis dans le code VHDL aux broches physiques de la carte.

Voici comment les signaux de l'encodeur PmodENC sont configurés dans le fichier de contrainte :

```
set_property PACKAGE_PIN V12 [get_ports {pmod[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {pmod[0]}]
set_property PACKAGE_PIN W16 [get_ports {pmod[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {pmod[1]}]
set_property PACKAGE_PIN J15 [get_ports {pmod[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {pmod[2]}]
set_property PACKAGE_PIN H15 [get_ports {pmod[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {pmod[3]}]
```

Listing 1: Configuration des ports pour le Pmod

Dans ces lignes de configuration, chaque signal de l'encodeur est attribué à une broche spécifique de la FPGA. Par exemple, `pmod[0]` est connecté à la broche physique `V12` de la carte FPGA, et ainsi de suite pour les autres signaux `pmod[1]`, `pmod[2]`, et `pmod[3]`. La directive `IOSTANDARD LVCMOS33` spécifie que les signaux sont configurés pour utiliser le standard logique LVCMOS33, qui est couramment utilisé pour les périphériques Pmod.

Ce travail nous a permis de comprendre le fonctionnement de l'encodeur rotatif et à poser les bases pour la création d'un bloc IP dédié, ce qui nous permettra d'intégrer cet encodeur dans des conceptions plus complexes à l'avenir.

4.2.2 AXI-enabled IP

Pour répondre à nos besoins avec le PmodENC, nous avons décidé de développer notre propre bloc IP. L'IP officielle proposée par Digilent ne convenait pas à notre application car elle nécessitait de coder le compteur 4 bits et la logique d'anti-rebond côté PS plutôt que côté PL. Notre objectif était d'intégrer

toute la logique de contrôle et de comptage dans la partie PL pour une meilleure performance et un contrôle plus direct.

Conception du Bloc IP Le bloc IP que nous avons créé, appelé `PmodENC_v1`, est composé de plusieurs composants :

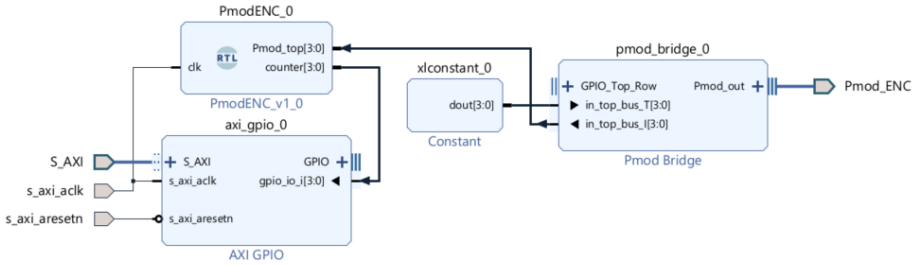

FIGURE 3 – Bloc IP

1. **AXI GPIO** : Ce bloc permet d'accéder aux registres GPIO via le bus AXI. Il est utilisé pour interagir avec le compteur et l'encodeur rotatif, permettant ainsi au processeur de lire et de manipuler les valeurs du compteur.
2. **Module RTL du Compteur** : Nous avons développé un module RTL en VHDL pour implémenter un compteur 4 bits intégré avec un mécanisme d'anti-rebond. Ce module traite directement les signaux de l'encodeur, comptabilise les impulsions et applique un filtrage pour éliminer les rebonds qui pourraient causer des erreurs de comptage.
3. **Pmod Bridge** : Ce composant sert d'interface entre le module PmodENC et la logique interne du FPGA. Il facilite la connexion des signaux du PmodENC aux broches GPIO appropriées sur la carte Zybo Z7-20, garantissant que les signaux sont correctement acheminés vers le compteur et traités adéquatement.
4. **Constant** : Ce bloc configure le tri-state dans le Pmod Bridge en définissant les broches du module PmodENC comme entrées ou sorties, selon les besoins du système.

Processus de Création du Bloc IP Le processus de création du bloc IP a commencé par la conception du module RTL du compteur en VHDL, voir section précédente.

Ensuite, nous avons intégré ce module dans un design plus large en utilisant Vivado. Nous avons ajouté le bloc AXI GPIO pour permettre la communication entre le processeur et notre module RTL via le bus AXI. Cette intégration a été facilitée par l'utilisation de l'assistant de création de bloc IP de Vivado, qui nous a permis de définir les interfaces AXI nécessaires.

Enfin, le Pmod Bridge a été configuré pour relier le PmodENC au module AXI GPIO, et les signaux ont été correctement mappés aux broches physiques de la carte Zybo Z7-20 en utilisant un fichier de contrainte XDC.

L'ensemble du design a été simulé et testé dans Vivado pour vérifier son fonctionnement correct avant d'être déployé sur la carte FPGA. Grâce à cette approche, nous avons pu créer un bloc IP personnalisé, intégrant toute la logique de traitement nécessaire directement dans le matériel, sans dépendre du système de traitement pour le comptage ou le filtrage des signaux.

Ce bloc IP constitue la base de notre implémentation Linux.

4.2.3 Baremetal

Ce projet, bien que facultatif, nous a permis de nous initier à l'utilisation de Vitis, l'environnement de développement intégré de Xilinx. Vitis ayant récemment subi une mise à jour de son interface, encore peu documentée officiellement, ce projet nous a permis de nous familiariser avec ses nouvelles fonctionnalités. L'objectif était de contrôler l'encodeur rotatif en développant une application baremetal en C, avec l'encodeur connecté au port JE de la carte Zybo Z7-20. Cette démarche visait à nous faire comprendre le processus de réalisation d'une application baremetal.

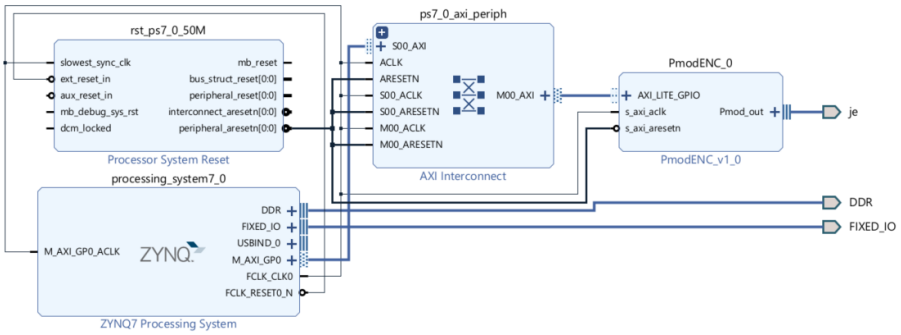

FIGURE 4 – Block design

Pour commencer, nous avons utilisé Vivado pour générer le matériel nécessaire. La première étape a été de créer un projet Vivado en sélectionnant la carte Zybo Z7-20. Ensuite, nous avons créé un BD où nous avons ajouté les composants nécessaires à notre application, incluant le processeur Zynq, qui constitue le cœur de la carte Zybo Z7-20, et le module PmodENC que nous avions précédemment réalisé. Une fois ces composants ajoutés, nous avons utilisé la fonctionnalité de connexion automatique de Vivado pour configurer correctement les liaisons entre les différents blocs. Avec le BD finalisé et les connexions établies, nous avons généré le fichier XSA à partir de Vivado. Ce fichier est requis par

Vitis, car il sert de base pour générer la plateforme sur laquelle l'application C sera compilée.

Après la création du fichier XSA, nous sommes passés à Vitis pour développer l'application baremetal en C. Nous avons importé le fichier XSA dans Vitis, ce qui nous a permis de configurer l'environnement logiciel en adéquation avec le matériel défini dans Vivado. Nous avons ensuite créé un nouveau projet de type "Application Project" en sélectionnant un template baremetal, qui permet de développer une application sans système d'exploitation, nous permettant ainsi de travailler directement avec le matériel.

Dans le code C de l'application, nous avons écrit les fonctions nécessaires pour initialiser le module PmodENC et lire les signaux de l'encodeur rotatif. Pour assurer une gestion efficace des changements d'état des signaux A et B, nous avons utilisé des interruptions, ce qui permet au système de réagir en conséquence de la rotation de l'encodeur.

Enfin, après avoir compilé l'application, nous l'avons téléchargée sur la carte Zybo Z7-20, ce qui nous a permis de tester et de vérifier le bon fonctionnement de notre code.

5 Reprogrammation dynamique

Nous disposons désormais d'une carte Zybo Z7-20 avec un RFS Linux personnalisé et de notre bloc IP faisant fonctionner le PmodENC en tandem du bus AXI. Il nous reste plus qu'à réaliser la reprogrammation dynamique et de enfin réaliser l'accélération matérielle.

5.1 FPGA Manager

Le FPGA Manager est une interface disponible dans le kernel Linux qui permet de charger et gérer des bitstreams sur un FPGA accessible par le système en exposant des fichiers virtuels par le biais du système de fichiers Linux. Ces fichiers sont accessibles à tout moment par l'utilisateur root, nous laissant ainsi la possibilité de faire notre reprogrammation dynamique.

Par ailleurs, cette interface peut être configurée pour des bitstreams complets ou partiels selon un système de flags mis en place par le FPGA Manager. Toutefois, ces bitstreams doivent passer par une étape supplémentaire si l'on veut les utiliser avec le FPGA Manager. De base, ils se présentent sous la forme de fichiers BIT qui sont utilisables par Vivado, mais l'interface nécessite des fichiers sous format BIN générés par l'outil `bootgen`. Ces fichiers binaires contiennent plus d'informations que ceux mentionnés précédemment et peuvent être utilisés pour faire des images entière de boot, toutefois ici nous utilisons juste `bootgen` comme moyen d'encapsuler le bitstream afin de le rendre utilisable par le FPGA Manager.

Sur PetaLinux, le FPGA Manager peut être activé de base dans la configuration du kernel ou peut ne pas l'être, il faut donc s'assurer que cette option doit bien être présente, les aspects techniques de cette opération sont détaillés dans notre documentation.

5.2 Block Design

Pour reprogrammer dynamiquement la partie PL, il nous faut donc un bitstream généré par Vivado puis transformé par bootgen. Or, celui que nous générons sera forcément un bitstream complet et non partiel vu que le procédé pour générer des bitstreams partiels nécessite un workflow différent. Il faut donc prendre en compte le fait que le bitstream généré doit à la fois contenir l'état de base de notre système et les nouvelles additions nécessaires au fonctionnement du PmodENC.

Il nous suffit donc de reprendre le BD utilisé pour mettre en place le RFS Linux et d'y rajouter notre bloc IP créé pour le PmodENC, les connexions entre le PS et le bloc IP peuvent être réalisées automatiquement par le Wizard de Vivado, facilitant ainsi l'implémentation.

Une fois fait, nous pouvons générer le bitstream et le transformer en un fichier BIN pour être utilisé dans les prochaines étapes.

5.3 Sans Device Tree Overlay

Nous avons désormais tous les éléments nécessaires pour réaliser un premier essai de reprogrammation dynamique. D'abord il nous faut transférer notre fichier BIN sur la Zybo, typiquement par simple transfert sur la carte SD ou par SSH.

Ensuite vient les manipulations sur les fichiers virtuels du FPGA Manager. Dans notre cas, tous ces fichiers se trouvent dans `/sys/class/fpga_manager/fpga0/` et tout accès doit être réalisé en tant que root. Pour la première modification, un "0" doit être rajouté dans le fichier `flags` présent dans le répertoire mentionné ci-dessus pour signifier l'utilisation d'un bitstream complet, un simple `echo 0 > flags` suffit.

Après la mise à 0 de ce fichier, notre fichier BIN doit être transféré à un endroit spécifique pour qu'il soit reconnu, en l'occurrence nous devons le placer dans le répertoire `/lib/firmware/`. Il se peut que ce répertoire ne soit pas créé de base, un `mkdir` peut donc être nécessaire. Une fois le fichier BIN transféré, nous pouvons revenir à notre chemin de base contenant les fichiers virtuels et rajouter le nom de notre fichier BIN dans le fichier `firmware`, ce sera la dernière étape.

Supposons que notre fichier se nomme `project.bit.bin`, un `echo "project.bit.bin" > firmware` sera suffisant. Cette opération, si le fichier est bien présent dans `/lib/firmware/`, reprogrammera la partie PL avec le bitstream donné et le BD réalisé sera fonctionnel. L'ensemble des commandes réalisées se trouve ci-dessous.

```
sudo -i
echo 0 > /sys/class/fpga_manager/fpga0/flags
mkdir -p /lib/firmware
cp project.bit.bin /lib/firmware/
echo project.bit.bin > /sys/class/fpga_manager/fpga0/firmware
```

Listing 2: Commandes pour reprogrammation sans DTO

Toutefois, durant notre projet, cette reprogrammation n'a pas fonctionné de la manière attendue. En effet, même si la partie PL était en marche et qu'un simple test avec les LEDs de la Zybo pouvait être réalisé avec succès, la partie PS semblait être dans un état de gel. Ou du moins, l'affichage du terminal semblait l'être, toute tentative de taper une commande ne donnait aucun retour visuel à l'écran. Seulement, une fois la partie PL mise à zéro par le bouton "RESET" présent sur la carte, l'affichage rattrape le retard et affiche tout ce qui a été réalisé depuis le gel. Ce problème n'a pas pu être résolu directement sans l'utilisation d'un autre outil décrit dans la section suivante.

Par ailleurs, même si notre BD remplit son rôle, nous ne pouvons pas récupérer les données de l'encodeur et son compteur depuis Linux étant donné que

le système ne connaît pas l'existence du PmodENC. Il nous faut donc trouver une autre solution.

5.4 Avec Device Tree Overlay

La partie PS connaît l'existence des différents registres et périphériques grâce au Device Tree générée par PetaLinux, or notre PmodENC passant par bus AXI ne s'y trouve pas étant donnée qu'il n'est pas un périphérique "de base". De plus, notre objectif est de faire en sorte que l'encodeur soit plus ou moins échangeable à chaud avec la reprogrammation, il nous est donc inutile de le mettre de base dans le DT.

Heureusement, il existe un moyen de modifier dynamiquement un DT, les Device Tree Overlays. Ces morceaux de DT permettent de modifier, ajouter ou même supprimer des éléments et des périphériques du système. Il est donc possible de créer un DTO consistant en un simple ajout de notre PmodENC, rendant ainsi possible une communication entre PS et PL. Par ailleurs, le FPGA Manager supporte les overlays, l'intégration est donc simple une fois l'overlay écrit.

Voici un équivalent de l'overlay réalisé à l'occasion de ce projet :

```
/dts-v1/;
/plugin/;

&fpga_full {
    firmware-name = "enc_wrapper.bit.bin";
};

&amba {
    PetaENC_0: PetaENC@40000000 {
        clock-names = "s_axi_aclk";
        clocks = <&clkc 15>;
        compatible = "generic-uio";
        reg = <0x40000000 0x1000>;
    };
};
```

Listing 3: DTO utilisé pour le PmodENC

Cet overlay est écrit avec la nouvelle syntaxe, en effet il existe une plus ancienne. Dans notre projet, nous avons utilisé l'ancienne mais les deux sont équivalentes et peuvent être utilisées interchangeablement.

Nous nous concentrerons surtout sur le contenu plutôt que la syntaxe, la première partie avec le `firmware-name` nous permet de spécifier le nom du fichier BIN à utiliser pour la reprogrammation, il nous sera donc plus nécessaire de spécifier ce nom de fichier dans les commandes de reprogrammation.

Vient ensuite la partie qui nous intéresse, celle du PmodENC. Grâce au fait que le bloc IP que nous avons créé plus tôt ait des capacités AXI nous permet de le lier en mémoire avec le PS. Nous faisons donc savoir par le DTO son adresse mémoire et l'espace qu'il occupe, en l'occurrence `0x40000000` et `0x1000` respectivement. `clocks` permet quant à lui de spécifier l'horloge à utiliser.

La ligne `compatible` spécifie toutefois quelque chose qui n'a pas été abordé jusque là : le driver à utiliser. En effet, le kernel Linux communique avec les composants physiques par des pilotes, des drivers, et le PmodENC en nécessite donc un aussi. Heureusement, il existe plusieurs drivers génériques nous permettant ainsi d'interagir rapidement avec l'encodeur et en écourtant le développement. Nous aurions dû écrire un driver pour gérer tout cela sinon.

Le driver retenu pour notre encodeur est UIO, nous permettant de créer des applications dans l'user space pouvant accéder à la zone mémoire spécifiée par le DTO. La propriété `compatible` est donc paramétrée avec `generic-uio` afin de faire savoir au système d'utiliser ce driver lorsque l'encodeur est rajouté par la reprogrammation.

Le DTO doit passer par une étape de compilation avec l'outil `DTC` afin de le transformer en fichier DTBO, un format binaire que le kernel Linux peut comprendre. En supposant que notre DTO a pour nom `pl.dtso`, exécuter cette commande permet de lancer la compilation :

```
\acrshort{DTC} -O dtb -o pl.dtbo -b 0 -@ pl.dtso
```

Notez cependant que les DTO et le driver UIO doivent être activés dans la configuration du kernel de PetaLinux pour que la procédure de reprogrammation avec DTO soit fonctionnelle. Etant donné que nous utilisons un RFS Linux personnalisé, il nous faut transférer les drivers générés par PetaLinux manuellement vers notre système.

Tout comme la procédure sans DTO, un "0" doit être rajouté à `flags` dans `/sys/class/fpga_manager/fpga0/` et le bitstream en BIN dans `/lib/firmware/`. Une fois cela fait, il faut aussi transférer le DTBO dans le même répertoire que le bitstream.

Enfin, il faut que l'overlay soit appliqué au système. Pour cela, nous pouvons utiliser `configfs`, un système de fichiers virtuel nous permettant de manipuler le kernel. Les commandes à faire sont les suivantes :

```
mkdir /configfs
mount -t configfs configfs /configfs
```

Listing 4: Montage de configfs

Grâce à cela, nous avons un chemin avec lequel nous pouvons appliquer notre overlay. En étant dans `/configfs/device-tree/overlays/`, il faut créer

un nouveau répertoire nommé `full` où dans un fichier `path` nous spécifions le nom de notre overlay, ici `pl.dtbo` avec `echo -n "pl.dtbo" > full/path`.

L'ensemble des commandes réalisées pour l'application sont donc les suivantes :

```
sudo -i
echo 0 > /sys/class/fpga_manager/fpga0/flags
mkdir -p /lib/firmware
cp project.bit.bin /lib/firmware/project.bit.bin
cp pl.dtbo /lib/firmware/
mkdir /configfs
mount -t configfs configfs /configfs
cd /configfs/device-tree/overlays/
mkdir full
echo -n "pl.dtbo" > full/path
```

Listing 5: Commandes pour reprogrammation avec DTO

Une fois tout cela fait, la reprogrammation est terminée. Contrairement à celle sans DTO, nous pouvons continuer à utiliser le terminal et la continuité est donc assurée. Il nous reste plus qu'à extraire les données du compteur de notre PmodENC.

Avec UIO, nous pouvons rapidement vérifier si le périphérique est bien reconnu en allant dans le répertoire `/dev/`, un fichier `uio0` devrait s'y trouver. Le "0" de `uio0` peut être un autre chiffre selon le nombre de périphériques UIO actuellement utilisés, mais en l'occurrence nous n'en avons qu'un.

Afin de voir la valeur actuelle du compteur, nous avons créé une simple application en C compilée par un cross-compiler GCC. Son code est disponible dans l'annexe A.

Grâce à tout cela, nous avons donc enfin une accélération matérielle de l'encodeur : la partie PL s'occupe entièrement de l'encodeur, de l'anti-rebond et du comptage tandis que la partie PS se contente de récupérer les données afin de pouvoir les utiliser pour un éventuel traitement ou programme.

En plus de cela, nous avons aussi rajouter le système d'interruptions à notre encodeur. Avec quelques modifications du BD, nous pouvons utiliser les capacités d'interruptions du PS afin de nous informer d'un changement de la valeur. Notre application réalisait auparavant qu'un simple polling, mais avec les interruptions nous pouvons afficher la valeur du compteur dès qu'un changement est détecté. Bien sûr dans le cadre de notre application ce n'est qu'une petite optimisation, mais pour une application plus lourde, cela évite de devoir dédier des ressources inutilement.

6 Conclusion

Globalement, tous les objectifs de ce projet TER ont été réalisés. En effet, nous sommes arrivés à faire fonctionner l'accélération matérielle entre PL et PS pour l'encodeur tout en ayant un RFS Linux personnalisé et surtout une documentation exhaustive. De ce fait, n'importe qui suivant cette dernière est capable de reproduire ce que nous avons fait à l'occasion de ce TER. Que ce soit la mise en place du Linux, la création du bloc IP et du BD correspondant, ou encore la reprogrammation dynamique avec DTO, toutes ces étapes sont détaillées.

6.1 Perspectives

Cependant, il y a toujours moyen d'aller plus loin et certaines améliorations peuvent être apportées au projet. En effet, nous nous sommes contentés d'utiliser un bitstream complet dans le cadre de ce projet, mais nous aurions très bien pu partir sur un workflow DFX afin de créer un bitstream partiel. Cela a l'avantage de réduire la taille du fichier BIN final, étant donné que nous n'aurions plus besoin d'encapsuler l'entièreté du BD à chaque fois que nous voulons rajouter un périphérique.

Par ailleurs, nous n'avons pas réellement abordé le sujet de boot à distance, ce qui aurait permis de ne plus devoir utiliser une carte SD mais juste la mémoire QSPI. En flashant le FSBL et U-Boot sur cette mémoire, ce dernier est capable de boot un kernel et un RFS à distance, découplant ainsi la Zybo Z7-20 du Linux, cela revient donc à faire un NFS Boot.

6.2 Difficultés rencontrées

Néanmoins, le projet s'est passé non sans accroches. La première piste que nous avons explorée au début du projet avec les outils open-source Xilinx s'est révélée beaucoup trop fastidieuse pour ce TER. C'est pourquoi nous avons utilisé PetaLinux la majorité du temps.

Faire fonctionner la partie UIO a aussi pris un certain moment à cause de simples oublis ou erreurs de notre part. Par exemple, nous avons fait de nombreuses tentatives avec ce driver sans que le module kernel était présent, ce qui forcément pose un certain problème. C'est alors que nous avons compris que PetaLinux empaquette les drivers avec son RFS à lui, et qu'il fallait donc les transférer manuellement.

C'est aussi avec l'UIO que nous avons remarqué une grave erreur dans notre bloc IP : les registres tri-state n'étaient pas mis en mode entrée du côté de l'encodeur. Il ne se passait donc rien lorsque nous l'utilisions.

6.3 Compétences développées

Somme toute, nous avons pu explorer en profondeur certains aspects des FPGA que nous n'avions alors jusque là jamais abordés comme les BD et le Linux embarqué. D'ailleurs, c'est avec ce dernier que nous avons pu découvrir le domaine des drivers, des modules kernel et du kernel en lui-même. Nous sommes donc capables de manipuler Vivado avec plus d'aisance qu'auparavant, d'utiliser PetaLinux pour mettre en place Linux sur une carte Zynq, d'écrire des Device Tree Overlay pour une reprogrammation dynamique et enfin d'utiliser le driver UIO pour des applications destinées à des périphériques génériques.

Références

[1] eeWiki. *Sources de RFS Debian*. Sources de RFS Debian "minimal" pour utilisation sur des systèmes embarqués tournant sur ARM. URL : `https://rcn-ee.com/rootfs/eewiki/minfs/`.

[2] Dwayne HERZBERG et Oleg PAILLOT. *Documentation TER*. Documentation réalisée à l'occasion de ce TER. URL : `https://ter-zybo.github.io/Documentations/`.

[3] Dwayne HERZBERG et Oleg PAILLOT. *GitHub TER*. Organisation GitHub regroupant tout le travail réalisé à l'occasion de ce TER sous forme de plusieurs répertoires Git. URL : `https://github.com/TER-Zybo/`.

Glossaire

ARM Advanced RISC Machine. 11–14
AXI Advanced eXtensible Interface. 14, 16, 17, 19, 21, 22

BD Block Design. 17, 19, 20, 23–25
BIN binary file. 19–22
BIT bitstream file. 19

DFX Dynamic Function eXchange. 24
DT Device Tree. 21
DTB Device Tree Blob. 10
DTBO Device Tree Blob for Overlay. 22
DTC Device Tree Compiler. 22
DTO Device Tree Overlay. 21–24

FPGA Field-Programmable Gate Array. 2, 5, 9, 11, 12, 14–17, 19–21, 25
FSBL First Stage Bootloader. 10, 13, 24

GCC GNU Compiler Collection. 8, 23
GPIO Global Purpose Input/Output. 16, 17

IP Intellectual Property. 2, 12, 14–17, 19, 22, 24

PL Programmable Logic. 5, 14–16, 19–21, 23, 24
PS Processing System. 5, 15, 19–24

QSPI Quad Serial Peripheral Interface. 10, 24

RFS Root File System. 2, 6, 9–11, 13, 19, 22, 24
RTL Register Transfer Level. 16

SBC Single-Board Computer. 9, 13
SDK Source Developement Kit. 12
SoC Software-on-Chip. 5, 12
SSH Secure Shell. 20

UIO Userspace Input/Output. 22–25

VHDL VHSIC Hardware Description Language. 2, 15, 16
Vitis Vitis Unified Software Platform. 8, 17, 18
Vivado Vivado Design Suite. 2, 8, 11, 12, 16–19, 25

XDC Xilinx Design Constraints. 15, 17
XSA Xilinx Support Archive. 12, 17, 18
XSCT Xilinx Software Command-line Tool. 8

Table des figures

Liste des tableaux

Liste des blocs de code

A Code de l'application UIO en C

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
#include <signal.h>
#include <stdbool.h>
#include <stdint.h>
#include <string.h>
#include <poll.h>

#define GPIO_GLOBAL_IRQ 0x11C
#define GPIO_IRQ_CONTROL 0x128
#define GPIO_IRQ_STATUS 0x120

void print_usage(const char *program_name) {
    printf("Usage: %s <uio_device_number> <address_offset>
    ↪  <map_size_in_hex> [polling|interrupt]\n", program_name);
}

volatile bool keep_running = true;

void int_handler(int dummy) {
    keep_running = false;
}

int main(int argc, char *argv[]) {
    if (argc < 4 || argc > 5) {
        print_usage(argv[0]);
        return EXIT_FAILURE;
    }

    int uio_device_number = atoi(argv[1]);
    off_t address_offset = strtoul(argv[2], NULL, 0);
    size_t map_size = strtoul(argv[3], NULL, 16);
    bool polling = (argc == 5 && strcmp(argv[4], "polling") ==
    ↪  0);
    bool interrupt = (argc == 5 && strcmp(argv[4], "interrupt")
    ↪  == 0);

    if (address_offset % sizeof(uint32_t) != 0) {
        fprintf(stderr, "Address offset must be aligned to 4
        ↪  bytes.\n");
        return EXIT_FAILURE;
```

```
}

char uio_device_path[64];
snprintf(uio_device_path, sizeof(uio_device_path),
↪  "/dev/uio%d", uio_device_number);

int uio_fd = open(uio_device_path, O_RDWR);
if (uio_fd < 0) {
    perror("Failed to open UIO device");
    return EXIT_FAILURE;
}

void *map_base = mmap(NULL, map_size, PROT_READ | PROT_WRITE,
↪  MAP_SHARED, uio_fd, 0);
if (map_base == MAP_FAILED) {
    perror("Failed to mmap");
    close(uio_fd);
    return EXIT_FAILURE;
}

struct pollfd uio_poll = {
    .fd = uio_fd,
    .events = POLLIN,
};

volatile uint32_t *addr = (volatile uint32_t *)((char
↪  *)map_base + address_offset);
volatile uint32_t *gier = (volatile uint32_t *)((char
↪  *)map_base + GPIO_GLOBAL_IRQ);
volatile uint32_t *ier = (volatile uint32_t *)((char
↪  *)map_base + GPIO_IRQ_CONTROL);
volatile uint32_t *isr = (volatile uint32_t *)((char
↪  *)map_base + GPIO_IRQ_STATUS);

int ret;

if (interrupt) {
    signal(SIGINT, int_handler);
    *gier = 0x80000000; //Global interrupt enable
    *ier = (uint32_t) 1; //Channel 1 interrupt enable
    uint32_t reenable = 1;
    printf("Printing value at address offset 0x%lx when an
    ↪  interrupt occurs. Press Ctrl+C to stop.\n",
    ↪  address_offset);
    while(keep_running) {
        ret = poll(&uio_poll,1,-1);
```

```
            if(ret >= 1) {
                read(uio_fd,&reenable,sizeof(uint32_t));
                uint32_t value = *addr;
                printf("Interrupt detected ! Value: 0x%x\n",
                ↪  value);
                if(*isr != (uint32_t) 0) {
                    *isr = (uint32_t) 1;
                }
                write(uio_fd,&reenable,sizeof(uint32_t));
                ↪  //Reenable the interrupt in the UIO subsystem
                ↪  of Linux, in theory shouldn't be needed but
                ↪  for some reason it is...
            }
        }

    } else if (polling) {
        signal(SIGINT, int_handler);
        printf("Polling value at address offset 0x%lx. Press
        ↪  Ctrl+C to stop.\n", address_offset);
        while (keep_running) {
            uint32_t value = *addr;
            printf("Value: 0x%x\n", value);
            sleep(1);
        }
    } else {
        uint32_t value = *addr;
        printf("Value at address offset 0x%lx: 0x%x\n",
        ↪  address_offset, value);
    }

    if (munmap(map_base, map_size) < 0) {
        perror("Failed to munmap");
    }

    close(uio_fd);

    return EXIT_SUCCESS;
}
```