

École Polytechnique de Montréal

INF3610 Laboratoire 4

Introduction à l'accélération matérielle par synthèse de haut niveau

Frédéric Fortier & Eva Terriault
02/11/2017

Table des matières

Introduction.....	2
Objectifs du laboratoire	2
I. Familiarisation avec la synthèse de haut niveau et Vivado HLS	3
1. Synthèse de haut niveau versus synthèse logique	3
Présentation rapide de Vivado HLS.....	4
Description de la plateforme à générer	8
Première séance : Travail à effectuer	10
1. Génération du projet Vivado pour affichage HDMI	10
Procédure	10
Copie en mémoire de la vidéo sur la carte	11
2. Simulation fonctionnelle : Implémentation du filtre de Sobel	11
Introduction à l'algorithme	11
Implémentation sur Vivado HLS.....	13
Questions initiales	13
Importation de votre code du lab 3	14
Exécution logicielle.....	14
II. Deuxième séance	15

Introduction

La conception de système sur puce ou sur FPGA peut être un processus long et coûteux. Dans un contexte où les temps de mise sur le marché sont de plus en plus court et la complexité des systèmes grandit, les méthodes traditionnelles de conception ont beaucoup de mal à suivre le rythme. Il est donc nécessaire de penser à de nouvelles méthodes de conception pour de tels systèmes. Parmi les solutions proposées pour résoudre (au moins partiellement) ce problème, la simulation à un haut niveau d'abstraction du système en cours de conception est presque universellement utilisée. En effet, avec des solutions telles que SystemC, il est possible de valider un design à différents niveaux d'abstraction, en commençant par une solution fonctionnelle, puis en raffinant la solution jusqu'à obtenir une solution synthétisable et donc pouvant être implémentée. Le fait de partir d'une bonne abstraction permet à la fois des simulations beaucoup plus rapides de systèmes complexe et permet de s'ajuster plus rapidement en cas de changement des requis vers le début de la conception du système.

Cependant, pour pouvoir passer facilement d'une solution fonctionnelle à une solution synthétisable, on doit aussi avoir les outils nécessaires pour limiter le plus possible la réécriture de code, surtout dans un autre langage : la version synthétisable ne devrait être qu'une version un peu plus près du matériel que du logiciel (par exemple, tenant compte qu'allouer dynamiquement de la mémoire est un non-sens pour décrire un module matériel, etc.). C'est ici qu'arrive la synthèse de haut niveau (SHN ou HLS pour *High-Level Synthesis* en bon français, parfois aussi appelée synthèse comportementale), qui fonctionne à un niveau d'abstraction plus élevée que la synthèse logique au niveau RTL vue jusqu'ici dans les cours INF1500 et INF3500 (ex. VHDL, Verilog) en prenant en entrée une description algorithmique dans un langage de haut niveau tel que SystemC ou C/C++.

Finalement, pour avoir le temps mise en marché le plus court possible, il est impératif de commencer le développement logiciel le plus tôt possible, sans attendre la fin de la conception matérielle du système. Encore une fois, l'abstraction permet d'avoir un modèle assez représentatif du système final assez rapidement.

Objectifs du laboratoire

Ce laboratoire présente une introduction à la conception de systèmes embarqués par accélération matérielle en langage de haut niveau, en développant un filtre de Sobel matériel sur FPGA, contrôlé par logiciel sur un processeur ARM Cortex A-9. Il vous sera demandé d'adapter votre code du laboratoire 3 (version UTF) afin de l'importer dans l'outil Vivado HLS, qui permet à la fois la simulation et la synthèse de haut-niveau. Une fois la simulation à haut niveau (C/C++) validée, vous devrez inclure votre filtre à un projet Vivado permettant la sortie d'images sur le port HDMI du Zedboard. Vous programmerez le processeur ARM pour qu'il envoie à votre module Sobel une vidéo noir et blanc non compressée à votre module pour que celui-ci affiche ensuite le résultat à l'écran. Finalement, vous aurez probablement à raffiner votre solution pour améliorer les performances de votre accélérateur.

Le lab se sépare donc en 3 parties distinctes, les deux premières indépendantes entre elles :

- Importer votre code du lab 3 dans Vivado HLS. Le simuler de manière fonctionnelle et vérifier qu'il est synthétisable.
- Créer une plateforme avec Vivado pour le Zedboard. S'assurer du fonctionnement de l'HDMI et lecture de la vidéo non compressée depuis la carte SD.
- Ajout du filtre à votre plateforme Vivado, test de performance, rétroaction si nécessaire¹.

L'objectif de ce laboratoire consiste à:

- Introduire l'étudiant à la conception de System on a Chip (SoC) à haut niveau d'abstraction,
- Initier l'étudiant à la synthèse de haut-niveau
- Estimer rapidement les ressources matérielles et la performance du design
- Valider un système avec une co-simulation logicielle/matérielle.

I. Familiarisation avec la synthèse de haut niveau et Vivado HLS

1. Synthèse de haut niveau versus synthèse logique

Nous avons vu (ou verrons bientôt) en classe qu'en déroulant les boucles, il est possible d'accélérer le calcul d'une opération sur processeur superscalaire comme le Cortex A9, à condition d'avoir un bon compilateur. Lorsque l'on porte une fonction du Cortex A9 vers du matériel (ici FPGA), il est également possible de procéder à cette opération de déroulement de boucles dans un but d'accélération. Le même principe s'applique au pipelining. Si la fonction à accélérer comporte peu de contrôle, pourquoi se limiter à 5 ou 7 (ou 19 sur un x86 Intel récent) étages? Si la fonction s'y prête bien, le gain est en général plus important que sur superscalaire. Pourquoi? C'est ce que nous allons observer dans la suite de cette section à l'aide de l'outil Vivado HLS de Xilinx.

Dans un premier temps nous allons rapidement² décrire ce qu'est la synthèse de haut niveau et la différencier de la synthèse logique (SL).

Vous avez expérimenté en INF1500 et INF3500 les produits d'Aldec (Active-HDL) qui permettent la synthèse logique³. Cette dernière est un processus par lequel une description du circuit, généralement au niveau de transfert de registre (RTL), est transformée en une mise en œuvre de la conception en termes de portes logiques, typiquement par un logiciel. Ces outils de

¹ Ça sera nécessaire.

² Une introduction sera aussi donnée dans le cours INF3610 (voir document le document Introduction à HLS Vivado (Section 3 du site Web) et des principes pour obtenir une bonne performance seront donnés dans ce lab, mais pour approfondir vos connaissances sur les principes de fonctionnement de la synthèse de haut niveau, suivre le cours INF8500.

³ Pour être plus exact, Active-HDL fournit un simulateur VHDL/Verilog, mais ne fait qu'appeler les outils de Xilinx pour la synthèse vers le FPGA.

synthèse (et bien d'autres) génèrent des flux binaires pour dispositifs logiques programmables tels que les FPGA, tandis que d'autres visent la création d'ASIC.

Alors que la synthèse logique utilise une description de niveau RTL, la synthèse de haut niveau fonctionne à un niveau d'abstraction plus élevé, à commencer par une description algorithmique dans un langage de haut niveau tel que SystemC ou C/C ++. La SHN est donc un processus de conception automatisé qui interprète une description algorithmique d'un comportement souhaité et crée le matériel numérique pour mettre en œuvre ce comportement. La sortie d'un outil SHN est généralement une description SystemC, Verilog ou VHDL au niveau RTL, qui peut ensuite être utilisée en entrée de la synthèse logique. La synthèse logique n'est donc pas appelé à disparaître, mais on risque de voir de plus en plus de SHN au cours des prochaines années dans le domaine de la conception des systèmes embarqués. (Vous pouvez comparer la synthèse logique et la SHN, respectivement à l'assembleur et au langage C++. L'assembleur fait toujours partie de la chaîne de compilation logiciel, mais le concepteur interagit d'abord avec le C++.)

Présentation rapide de Vivado HLS

Note : des explications plus détaillées de comment générer une architecture efficace seront données plus loin dans ce lab.

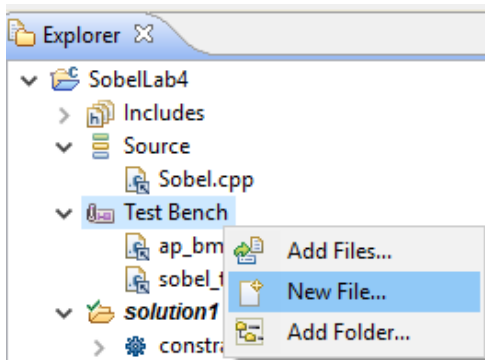
Note 2 : Allez aussi lire les slides [Introduction à HLS Vivado](#) sur le site du cours.

Note 3 : Pour de vrai. Allez lire les slides, le reste des explications de ce lab prend pour acquis que vous l'avez fait.

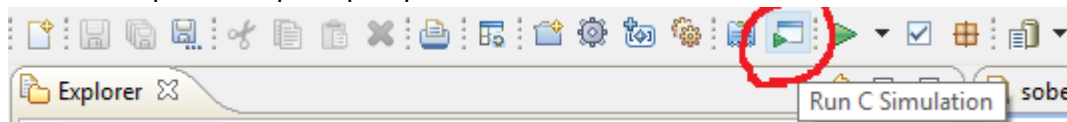
Vivado HLS est un logiciel de synthèse de haut niveau propriétaire de Xilinx, pour les FPGA Xilinx. Il prend en entrée un sous-ensemble du C/C++ ou un modèle SystemC et ressort du code VHDL et Verilog synthétisable. Le logiciel fonctionne sur un modèle centrée autour d'une IP (*Intellectual Property*, l'équivalent en design de puces d'un module ou d'une librairie en logiciel), permettant de la développer, de la tester et d'estimer ses performances dans la même interface graphique. Par contre, l'intégration à un système complet se fait au niveau de l'exportation de l'IP développée vers Vivado (vu au début du lab 2). Ainsi, une fois votre filtre de Sobel C/C++ transformé en VHDL, le nécessaire est automatiquement généré pour pouvoir l'ajouter à votre système, de la même manière que, par exemple, vous ajoutiez un contrôleur d'interruptions (AXI *Intc*) à votre système au lab 2.

L'interface graphique de Vivado HLS, encore une fois basée sur Eclipse permet :

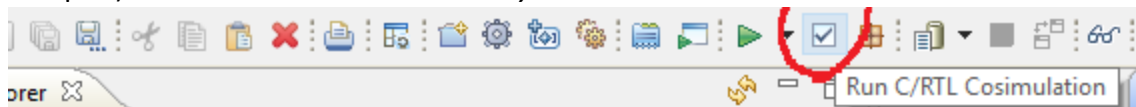
- De créer ou d'importer votre module à synthétiser, programmé en C/C++
- D'ajouter un banc d'essai pour tester votre module



- De simuler votre module avant de l'exporter pour vérifier son comportement
 - En compilant le banc d'essai et votre module comme un logiciel x86, ce qui permet de valider rapidement son fonctionnement. C'est l'équivalent du niveau de simulation UTF vu en classe et au lab 3. De plus, comme le module est exécuté nativement, la simulation prend au plus quelques secondes.

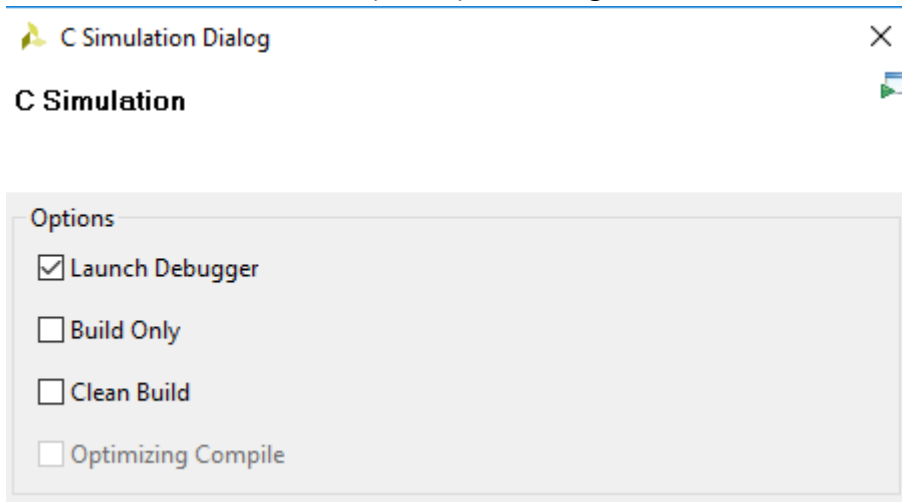


- En co-simulation logiciel/matériel : le banc d'essai est exécuté en logiciel, mais donne la main au simulateur VHDL de Xilinx lorsque vient le temps d'exécuter le module matériel. Cette simulation est alors exécutée sur le VHDL en sortie de la synthèse, et non sur le code C/C++ d'origine. Par contre, comme il s'agit alors d'un simulateur, l'exécution est beaucoup plus lente (10~25 minutes pour votre filtre de Sobel complet). Le résultat obtenu est alors *Cycle Accurate*⁴

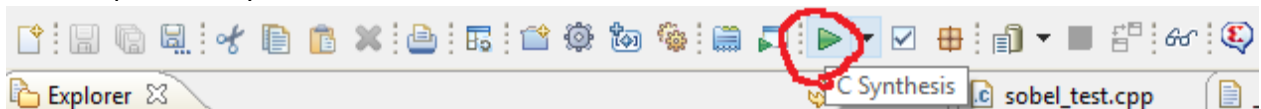


⁴ La simulation est *Cycle Accurate* au niveau du module, mais ne peut pas prendre en compte certains effets externes, comme la latence de communication avec la mémoire DDR où l'image de votre filtre se trouve. Le résultat final réel pourrait être (et sera probablement) plus lent.

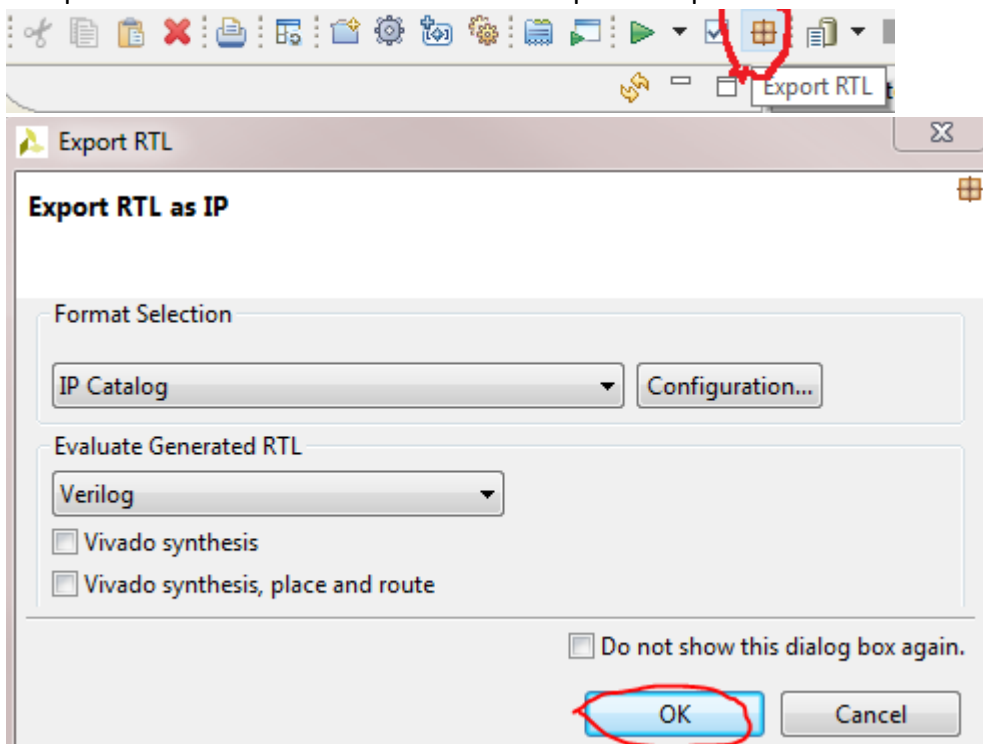
- En simulation fonctionnelle (C/C++), de déboguer votre module ou votre banc d'essai



- Sans surprise, de synthétiser votre module



- D'exporter votre module sous un format importable par Vivado



- D'obtenir un estimé de la fréquence, du nombre de cycles nécessaires à l'exécution et de la consommation de ressources du module synthétisé⁵.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	8.00	7.00	1.00

Ici le *target* d'horloge correspond au temps demandé à la création du projet. Le champ estimé correspond à l'horloge atteignable pour cette synthèse.

Latency (clock cycles)

Summary

	Latency		Interval		Type
	min	max	min	max	
	4169884	4169884	4169885	4169885	none

La latence correspond au nombre de cycles nécessaires pour exécuter le module

Detail

Instance

Loop

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
- first	1921	1921	3	1	1	1920	yes
- buffer_fill	3862	3862	1931	-	-	2	no
+ buffer_fill.1	1921	1921	3	1	1	1920	yes
- L1	4162158	4162158	3861	-	-	1078	no
+ L2	3846	3846	9	2	1	1920	yes
- last	1926	1926	8	1	1	1920	yes

Les informations sur les boucles contiennent leur temps d'exécution et les détails du pipeline généré.

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	1192	714
FIFO	-	-	-	-
Instance	4	-	1816	1496
Memory	4	-	0	0
Multiplexer	-	-	-	615
Register	-	-	920	96
Total	8	0	3928	2921
Available	280	220	106400	53200
Utilization (%)	2	0	3	5

Estimation des ressources consommées par le module

- D'avoir, si besoin, une analyse cycle par cycle des opérations effectuées et des accès aux différentes ressources (vue *Analysis* en haut à droite, pourrait être utile si vous cherchez à améliorer votre performance).

⁵ On n'est ici qu'en présence d'un estimé, car à ce stade-ci, seule la synthèse C/C++ vers VHDL a été réalisée. Le résultat exact final n'est connu qu'une fois la synthèse logique (VHDL vers binaire pour FPGA) est faite.

Description de la plateforme à générer

La plateforme à générer pour le lab est présentée visuellement à la page suivante. Les pointillés correspondent à des éléments sur FPGA et les traits pleins des éléments fixes du SoC Zynq. La partie FPGA comporte 3 éléments principaux :

Le *pipeline HDMI* : une suite de modules (IP) effectuant des transformations dans le but de transformer un tableau de pixels sous format RGBA (RGB + alpha, soit 4 octets/pixel) dans un format et à une vitesse acceptable pour le transfert au transmetteur HDMI, une puce externe qui se chargera par la suite de la conversion électrique des signaux pour le transfert sur le câble HDMI. Il n'est pas nécessaire ici d'entrer dans les détails de ce pipeline : il suffit de savoir que le tableau de pixels correspondant à l'image est simplement un tableau à la fin de la mémoire principale (le *framebuffer* en français), de taille 1920x1080x4 octets⁶. Ainsi, assigner 0 sur le premier élément (pixel de 4 octets) de ce tableau afficherait une couleur noire au premier pixel en haut à gauche de l'écran, assigner une valeur de 0xFFFFFFFF afficherait un pixel blanc.

Contrôleur I²C (IIC) : Fait également partie de la gestion de l'HDMI. C'est avec ce protocole que la carte envoie à l'écran la configuration de la résolution et autres paramètres nécessaires à un affichage correct. Encore une fois, une compréhension des détails n'est pas nécessaire au lab.

Filtre de Sobel : Présent à partir de la troisième partie du lab, c'est votre filtre synthétisé sur le FPGA. Il est chargé d'aller chercher en mémoire principale l'image à traiter (sous forme de pixel noir et blanc, donc 1 octet/pixel), et d'écrire en mémoire le résultat (sous forme de pixel RGBA, donc 4 octets/pixel).

Il est important de réaliser que le pipeline HDMI et le filtre de Sobel ont un accès direct en lecture et écriture à la mémoire principale (DDR extérieure à la puce), partagée avec le processeur ARM. La copie depuis et vers ces modules ne passe donc pas par le processeur ARM. Ce sont ces deux modules qui se chargent d'aller chercher les données qu'ils ont besoin, le cas échéant, d'écrire le résultat. Ces modules ont toutefois chacun un bloc de registres plus traditionnel accessibles par le processeur (via un bus AXI-Lite séparé⁷). Ces registres servent à indiquer à ces modules les adresses à accéder en mémoire pour aller, respectivement, lire le *framebuffer* et lire l'image filtrée et écrire le résultat.

Finalement, notez qu'une représentation plus exacte des chemins des données du Zynq est présentée en annexe de cet énoncé. Elle n'est cependant pas nécessaire à la compréhension de ce lab.

⁶ Pour une résolution de 1080p. Une résolution de 720p, par exemple, aurait un *framebuffer* de 1280x720x4 octets.

⁷ Les détails du bus AXI-Lite (et AXI) seront vus en classe bientôt. La compréhension de ces détails n'est cependant pas nécessaire pour faire ce laboratoire.

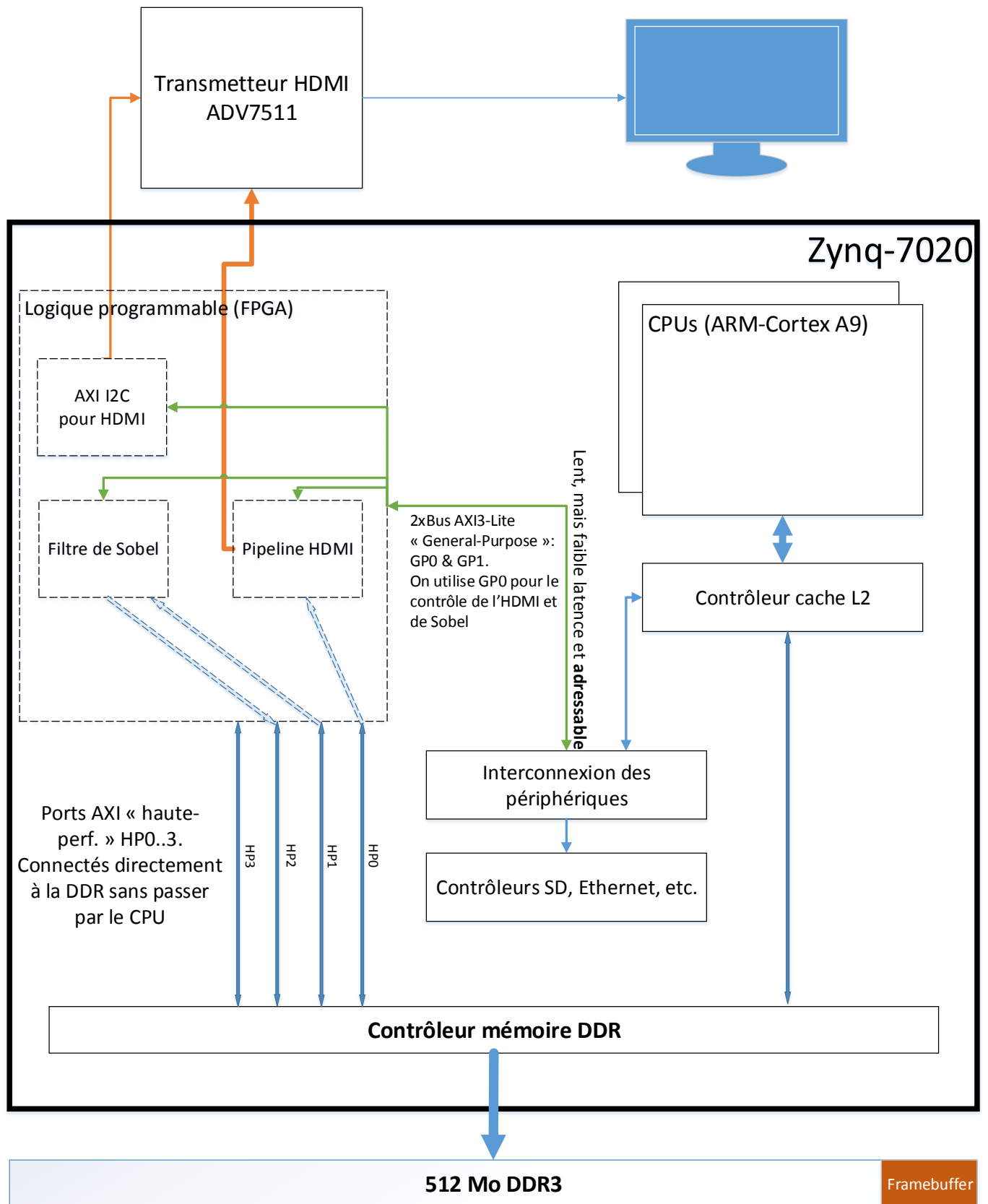


Diagramme simplifié des interconnexions du Zedboard

Première séance : Travail à effectuer

1. Génération du projet Vivado pour affichage HDMI

Procédure

Suivez la procédure *Projet Vivado avec HDMI* pour créer un projet Vivado et SDK, supportant l’affichage HDMI sur le Zedboard. Arrêtez à la fin de la deuxième section (ex. vous ne pouvez pas ajouter votre filtre de Sobel qui n’est pas encore fait).

Note : Pendant la génération du bitstream, vous pouvez faire la partie ci-bas avec ffmpeg pour sauver du temps.

Vidéo non compressée

Le filtre de Sobel que vous développerez travaille directement sur une vidéo non compressée/encodée, noire et blanc, à un octet/pixel. Cependant, les vidéos sont à peu près tout le temps encodées, puisqu’à 1920x1080 pixels/image x 30 images/seconde, on arrive rapidement à une quantité énorme de données. Nous nous limiterons donc pour ce laboratoire à une vidéo d’un maximum de 6 secondes (soit environ 355 Mo) pour qu’elle rentre au complet dans la mémoire vive de la carte.

Nous utiliserons donc l’utilitaire en ligne de commande [ffmpeg](#), qui peut être téléchargé [ici](#) pour convertir une vidéo encodée en son équivalent noir et blanc non compressé. La vidéo *amos9.mp4* fournie avec l’énoncé est utilisée ici, vous pouvez aussi en utiliser une autre tant que la résolution reste de 1920x1080. La commande

```
ffmpeg -i amos9.mp4 -t 00:00:06.00 -c:v rawvideo -pix_fmt gray a9.rgb
```

convertit les 6 premières secondes de la vidéo dans le fichier de sortie *amos9.rgb*. Notez que l’extension est trompeuse, puisque cette vidéo est en tons de gris à un octet/pixel, mais ffmpeg nous force à utiliser cette extension. Notez qu’il est aussi possible de faire la transformation inverse avec

```
ffmpeg -f rawvideo -s:v 1920x1080 -r 30 -pix_fmt gray -i a9.rgb -c:v h264 out.mp4
```

Finalement, il est suggéré de garder 2 versions de la vidéo, une de 6 secondes pour vous vanter de votre travail une fois celui-ci complété, et une version plus courte (une demie à une seconde) pour travailler, le temps de chargement de 355 Mo depuis une carte SD étant assez conséquent.

```
ffmpeg -i amos9.mp4 -ss 00:00:02.00 -t 00:00:01.00 -c:v rawvideo -pix_fmt gray a9s.rgb
```

Le paramètre additionnel *ss* commençant la conversion à la deuxième seconde.

Une fois la/les vidéos converties, copiez-les sur la racine de la carte SD. Essayez de donner des noms de 8 caractères ou moins à vos vidéos, ce qui vous évitera de modifier l’option de support de longs noms de fichiers dans le SDK.

Copie en mémoire de la vidéo sur la carte

On veut pouvoir copier l'extrait de vidéo convertie de la carte SD vers la mémoire de la carte. Pour ce faire, Xilinx fournit (sous le nom *xilffs* dans la génération du BSP) la librairie FAT32 pour systèmes embarqués *FatFs*. Le BSP s'occupant automatiquement de configurer le périphérique SDIO pour la lecture de la carte SD, il vous suffit d'utiliser [l'API de FatFs](#) pour lire les données de la carte SD. Les différentes fonctions accessibles à partir de l'en-tête *ff.h*, déjà incluse dans le code fourni. Il s'agit donc simplement d'implémenter la fonction `getFileContents()` dans le code fourni.

Une fois ceci fait, vous devriez voir votre extrait de vidéo à l'écran (avec un *framerate* plutôt mauvais). Notez que la lecture de la carte SD peut être assez lente, d'une dizaine de secondes pour une vidéo d'une seconde à 30 à 60 secondes pour une vidéo de 6 secondes.

2. Simulation fonctionnelle : Implémentation du filtre de Sobel

Introduction à l'algorithme

Note : Cette section est principalement une reprise de l'énoncé du lab 3 et, comme au lab 3, vous n'aurez pas besoin de comprendre les détails de cet algorithme, mais simplement d'adapter votre code du lab 3 (UTF). Vous aurez cependant peut-être à modifier cette implémentation pour améliorer les performances.

L'algorithme de Sobel est un algorithme couramment utilisé en traitement d'images pour extraire les contours de celle-ci. Il fonctionne en approximant la dérivée de l'image (les contours étant nécessairement aux endroits où cette dérivée est plus grande) à l'aide de deux tables pré-calculées de coefficients données au tableau 1.

-1	-2	-1
0	0	0
1	2	1

-1	0	1
-2	0	2
-1	0	1

Tableau 1 : Masques de coefficients de l'algorithme de Sobel

L'algorithme consiste donc à attribuer une valeur, pour chaque pixel de l'image, égale à la somme de la valeur des pixels avoisinants multipliés par le coefficient correspondant, ce qui permet de passer d'une image comme celle de la figure 11 à une détection de contours comme celle de la figure 12.

Les étudiants intéressés à avoir plus de détails sur l'algorithme ou le traitement d'image en général sont invités à consulter la page Wikipedia dédiée à celui-ci et/ou à suivre le cours INF4725 : Traitement de signaux et d'images, ce laboratoire ayant plutôt pour but de montrer le flot de conception d'une solution matérielle/logiciel à un problème donné.



Figure 1: Image originale




Figure 2: Résultat de l'application du filtre de Sobel sur l'image précédente

Implémentation sur Vivado HLS

Ouvrez Vivado HLS 2017.2, puis ouvrez le projet Sobellab4, préalablement copié dans `C:/TEMP/3610_4/matricule1_matricule2`.

Questions initiales⁸

Synthétisez  le début de code fourni. Notez la latence totale, les informations des deux boucles (latence, latence d'itération) et l'utilisation des ressources. Notez les définitions suivantes pour les métriques de boucles.

Les statistiques des sous-boucles sont incluses dans leur parent

Temps total d'exécution de la boucle

Temps pour une itération de la boucle

Si la boucle est pipelinée, nombre de cycles entre le début d'une itération et le début de la suivante

Loop

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
- IMG	18662400	18662400	9	-	-	2073600	no
+ OneTo4	4	4	1	1	1	4	yes

Les unités sont en cycles

Nb. d'itérations

Ajoutez, sous la boucle OneTo4, la ligne `#pragma HLS unroll`, puis resynthétisez et re-notez les mêmes métriques.

```
IMG: for (int i = 0; i < IMG_WIDTH * IMG_HEIGHT; ++i) {  
    uint8_t val = inter_pix[i];  
    OneToFourPixels fourWide;  
    OneTo4: for (int j = 0; j < 4; ++j)  
    #pragma HLS unroll  
        fourWide.pix[j] = val;  
    out_pix[i] = fourWide.full;  
}
```

Enlevez le pragma, puis répétez l'ajout de pragma, resynthèse et prise des métriques pour :

- `#pragma HLS unroll factor=8` sous la boucle IMG (8 car c'est la latence d'itération sans aucun pragma. Vous pouvez jouer avec le facteur par la suite si vous le désirez, mais vous verrez le temps de synthèse augmenter, mais les conclusions devraient rester similaires).
- `#pragma HLS pipeline` sous la boucle OneTo4.

⁸ À faire avant de modifier le code. Répondez à ces questions tout de suite, ou au minimum faites les sans répondre sur papier et gardez vos conclusions en tête pour la partie d'amélioration des performances de la deuxième séance.

- *#pragma HLS pipeline* sous la boucle IMG. Notez que le pipeline déroule implicitement les sous-boucles (OneTo4 dans ce cas).

Question 1. Vous avez un code impliquant une convolution 2D 3x3 (soit une multiplication-accumulation sur 2 boucles imbriquées à 3 itérations/boucle, comme *sobel_operator*) à accélérer par HLS. Cette convolution est exécutée assez souvent, mais pas de manière régulière (ex. pas dans une plus grosse boucle sans contrôle⁹). En vous basant sur les résultats précédents, devriez-vous dérouler ou pipeliner ces deux boucles? Expliquez pourquoi votre choix est le meilleur, et pourquoi l'autre choix est moins bon (autrement dit, donnez l'avantage de votre choix dans cette situation, et le désavantage de l'autre choix dans cette situation).

Question 2. Vous avez un code impliquant une convolution 2D 3x3 à accélérer, mais celle-ci est appelée régulièrement et sans contrôle significatif en boucle pour couvrir une image entière de résolution appréciable. En vous basant sur les résultats précédents, devriez-vous dérouler ou pipeliner cette boucle principale ? Expliquez pourquoi votre choix est le meilleur, et pourquoi l'autre choix est moins bon (autrement dit, donnez l'avantage de votre choix dans cette situation, et le désavantage de l'autre choix dans cette situation).

Importation de votre code du lab 3

Importez votre code (UTF) du filtre de Sobel du lab 3 dans le fichier Sobel.cpp. Les modifications/points suivants sont à faire/noter :

- Nous n'utiliserons pas SystemC pour ce lab. Il serait parfaitement possible de le faire, mais comme le lab ne comporte qu'un seul module et que la synthèse automatique d'interfaces est plus complexe avec SystemC (où, dans un vrai projet, il est entendu qu'un modèle de ces interfaces serait déjà disponible, ce qui n'est pas le cas ici), l'approche purement C a été préférée. Notez qu'il suffit ici d'appeler la fonction *sobel_filter()* en passant les bons paramètres.
- La taille de l'image n'est plus variable. Elle est *IMG_WIDTH * IMG_HEIGHT*, définies à *1920 * 1080*.
- *sobel_filter* prend en entrée des pixels noir et blanc (donc 1 octet/pixel, type *uint8_t*) mais ressort des pixels RGBA (4 octets/pixel, type *unsigned* c'est le format utilisé en entrée de l'affichage HDMI). **Chacune des couleurs (octets) de la sortie doit donc être affectée à la même valeur** (on quadruple donc l'information, comme dans le code initial fourni).
- N'oubliez pas de décommenter les pragma d'interface.
- Un banc de test est fourni, qui applique le filtre sur une image. Il vérifie ensuite l'exactitude du résultat. L'image résultante est disponible dans le dossier du projet (*result.bmp*).
- L'opérateur *new* n'est pas synthétisable.
- Les détails seront vus dans la troisième partie, mais les tableaux déclarés dans un module HLS sont synthétisés avec des blocs BRAM (ou des registres en FF/LUT si le tableau est très petit) sur le FPGA. Cette BRAM sert habituellement de cache personnalisée à l'algorithme implémenté. L'image entière à 1 octet/pixel occupe 1024 des 280 blocs BRAM disponibles.

⁹ Par contrôle on veut dire if/else qui déterminent si on appelle ou non la convolution.

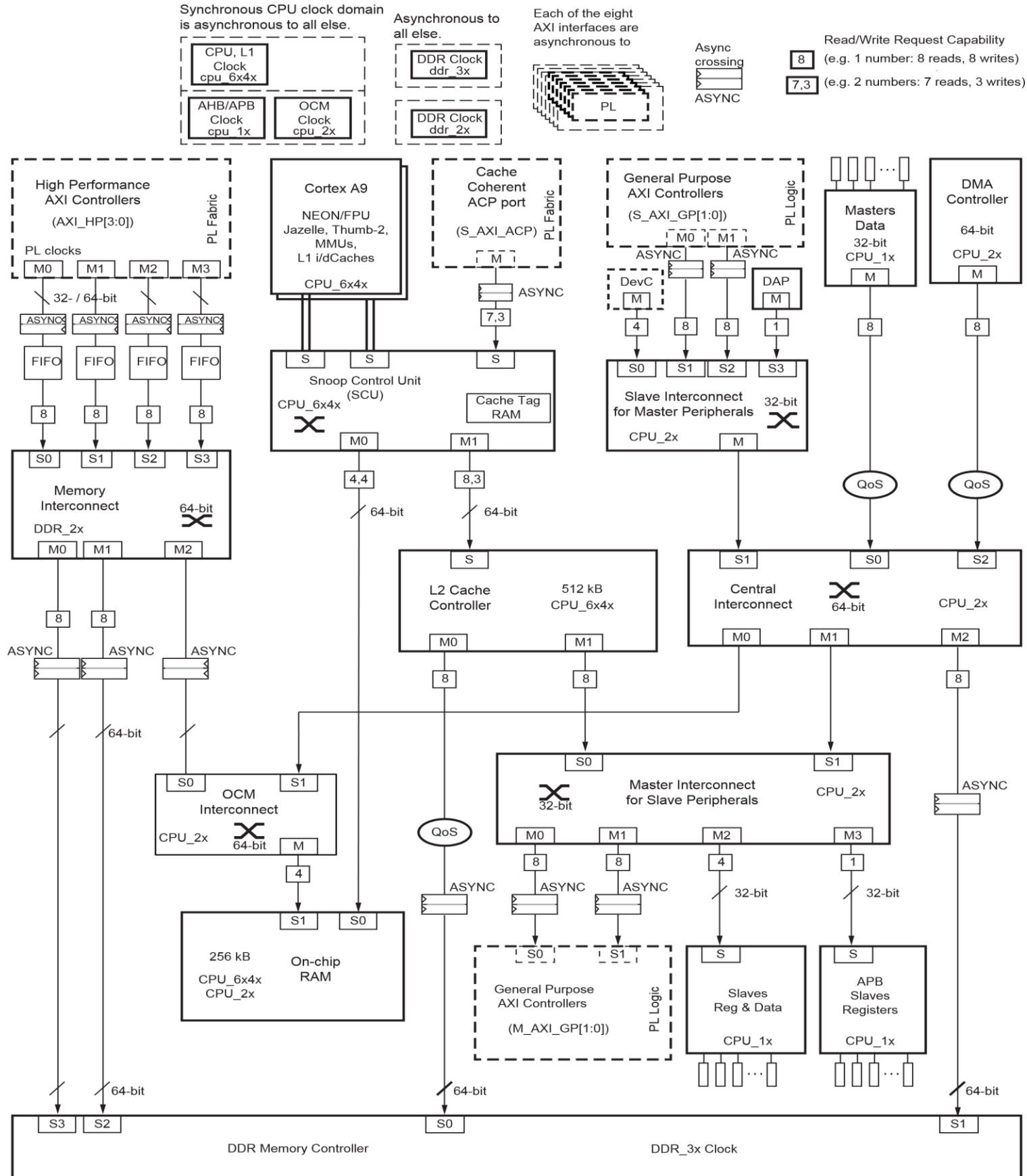
- Votre filtre devrait être synthétisable, et le nombre d'itérations des boucles et son délai connu et fixe (ex. pas une plage de valeurs possibles).

Exécution logicielle

Une fois votre code fonctionnel et en assez bon état, importez-le dans votre projet SDK en suivant la procédure *Projet Vivado avec HDMI* (si vous n'avez pas complété la Partie 1 - [Génération du projet Vivado pour affichage HDMI](#), faites-le préalablement). Décommentez l'appel à `sobel_filtrer()` dans `doSobelSW()` (et l'include au début du fichier) et modifiez votre code du main pour appeler `doSobelSW()`. **Notez la performance obtenue.**

II. Deuxième séance

À venir sous peu.



UG585_c5_01_120813

Interconnections du Zynq-7000. Xilinx, *Zynq-7000 All Programmable SoC Technical Reference Manual (UG585)*, adresse :

https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf