École Polytechnique de Montréal

INF3610 Laboratoire 4

Introduction à l'accélération matérielle par synthèse de haut niveau

Frédéric Fortier & Eva Terriault
08/11/2018

Table des matières

Introduction 2
Objectifs du laboratoire 2
I. Familiarisation avec la synthèse de haut niveau et Vivado HLS 3
Synthèse de haut niveau versus synthèse logique 3
Présentation rapide de Vivado HLS 4
II. Description de la plateforme à générer 8
III. Travail à effectuer 10
1. Simulation fonctionnelle : Implémentation du filtre de Sobel 10
Introduction à l'algorithme 10
Implémentation sur Vivado HLS 12
Étapes à suivre et questions 12
Importation de votre code du lab 3 13
2. Génération du projet Vivado pour affichage HDMI 13
Procédure 13
Vidéo non compressée 13
Copie de la vidéo sur la mémoire de la carte 14
Exécution logicielle 15
Exécution matérielle du filtre sur la carte 15
Barème de performance du filtre matériel 15
3. Amélioration de la performance du filtre matériel 16
Cache personnalisée pour le module 16
Transaction *burst* et flot des données 17
Déroulement de boucles 18
Pipelinage 19
Partitionnement de tableaux 19
Modification de la fréquence 19
Conseils généraux 20
Questions 20
IV. Procédure de remise 21
V. Barème 22

VI. Références.....22

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 en marché sont de plus en plus courts 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 anglais, 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 de 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 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 :

1. Importer votre code du lab 3 dans Vivado HLS. Le simuler de manière fonctionnelle et vérifier qu'il est synthétisable.
2. 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.
3. Ajout du filtre à votre plateforme Vivado, test de performance et rétroaction.

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

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 pipelinage. 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 rapidement[1] 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 logique[2]. 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

[1] 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 davantage vos connaissances sur les principes de fonctionnement de la synthèse de haut niveau, suivre le cours INF8500.

[2] 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.

œuvre de la conception en termes de portes logiques, typiquement par un logiciel. Ces outils de 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 disparaitre, 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 chaine de compilation logiciel, mais le concepteur interagit d'abord avec le C++.)

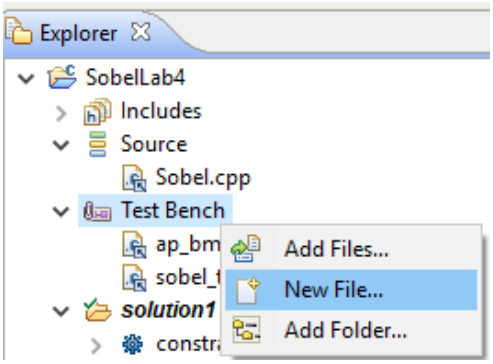
Présentation rapide de Vivado HLS

Note : des explications plus détaillées de comment générer une architecture efficace seront données plus loin dans ce labmais vous devez d'abord lire les slides Introduction à HLS Vivado sur le site du cours.

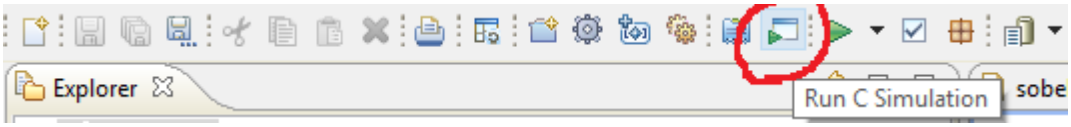
Vivado HLS est un logiciel de synthèse de haut niveau propriétaire de Xilinx, pour les FPGA Xilinx. Il prend en entrée un sous-ensemble du C/C++ ou un modèle SystemC et ressort du code VHDL et Verilog synthétisable. Le logiciel fonctionne sur un modèle centré autour d'une IP (*Intellectual Proprety*, 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 :

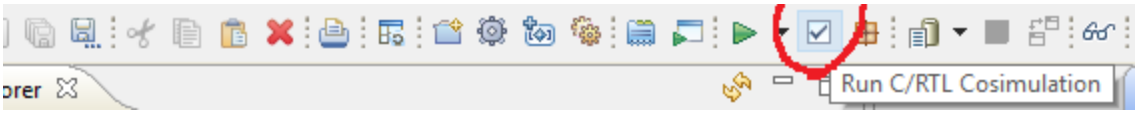
- De créer ou d'importer votre module à synthétiser, programmé en C/C++
- D'ajouter un banc d'essai pour tester votre module



- 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.

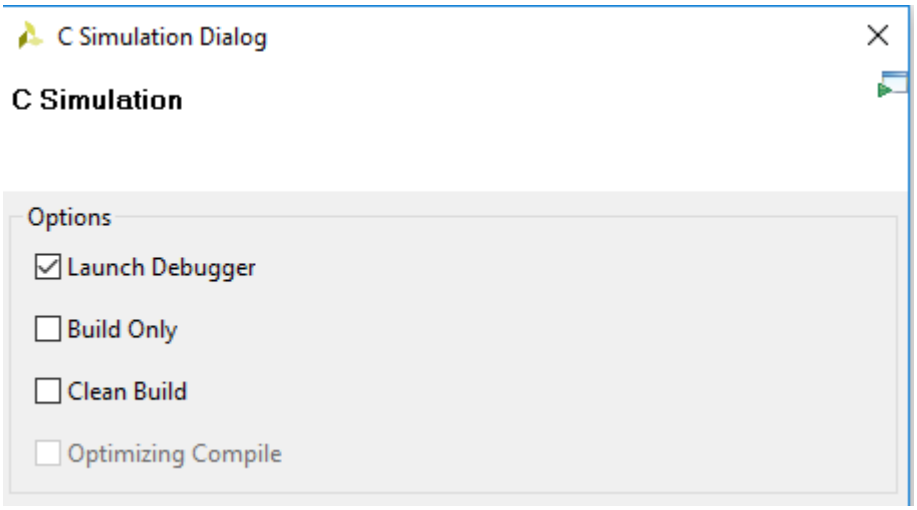


    - En co-simulation logiciel/matériel : le banc d'essai est exécuté en logiciel, mais donne la main au simulateur VHDL de Xilinx lorsque vient le temps d'exécuter le module matériel. Cette simulation est alors exécutée sur le VHDL en sortie de la synthèse, et non sur le code C/C++ d'origine. Par contre, comme il s'agit alors d'un simulateur, l'exécution est beaucoup plus lente (10~25 minutes pour votre filtre de Sobel complet). En effet, ici le niveau d'abstraction est *Cycle Accurate*[3]

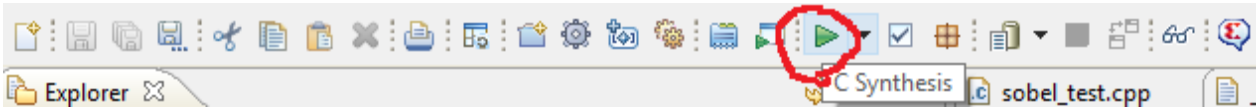


[3] La simulation est *Cycle Accurate* au niveau du module, mais ne peut pas prendre en compte certains effets externes, comme la latence de communication avec la mémoire DDR où l'image de votre filtre se trouve. Le résultat final réel pourrait être (et sera probablement) plus lent.

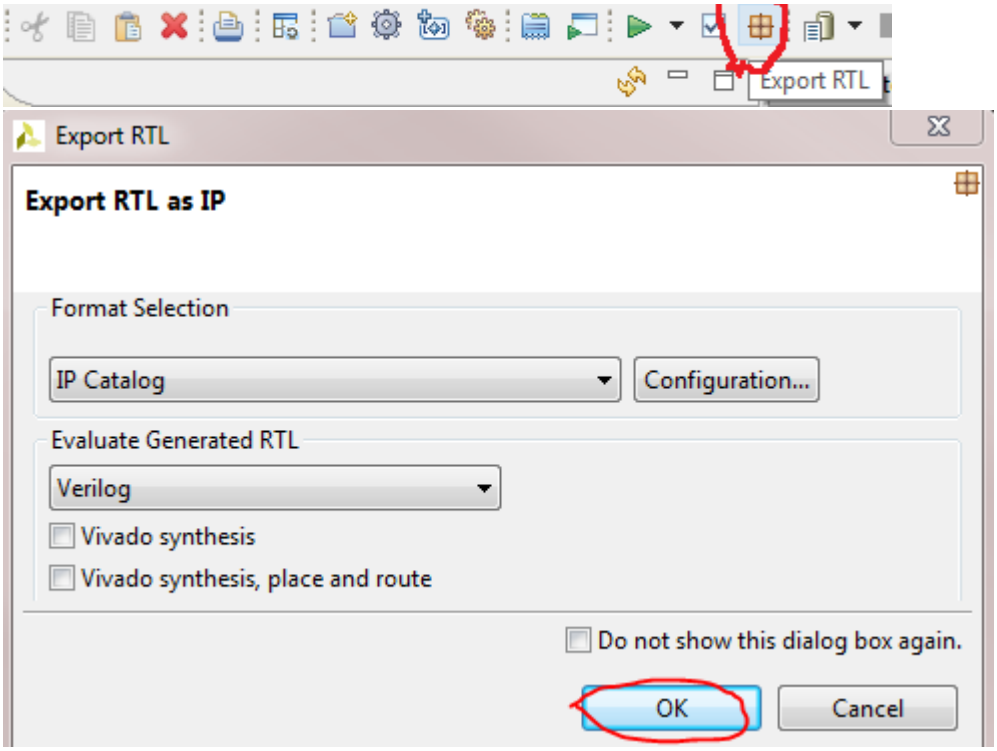
- En simulation fonctionnelle (C/C++), de déboguer votre module ou votre banc d'essai



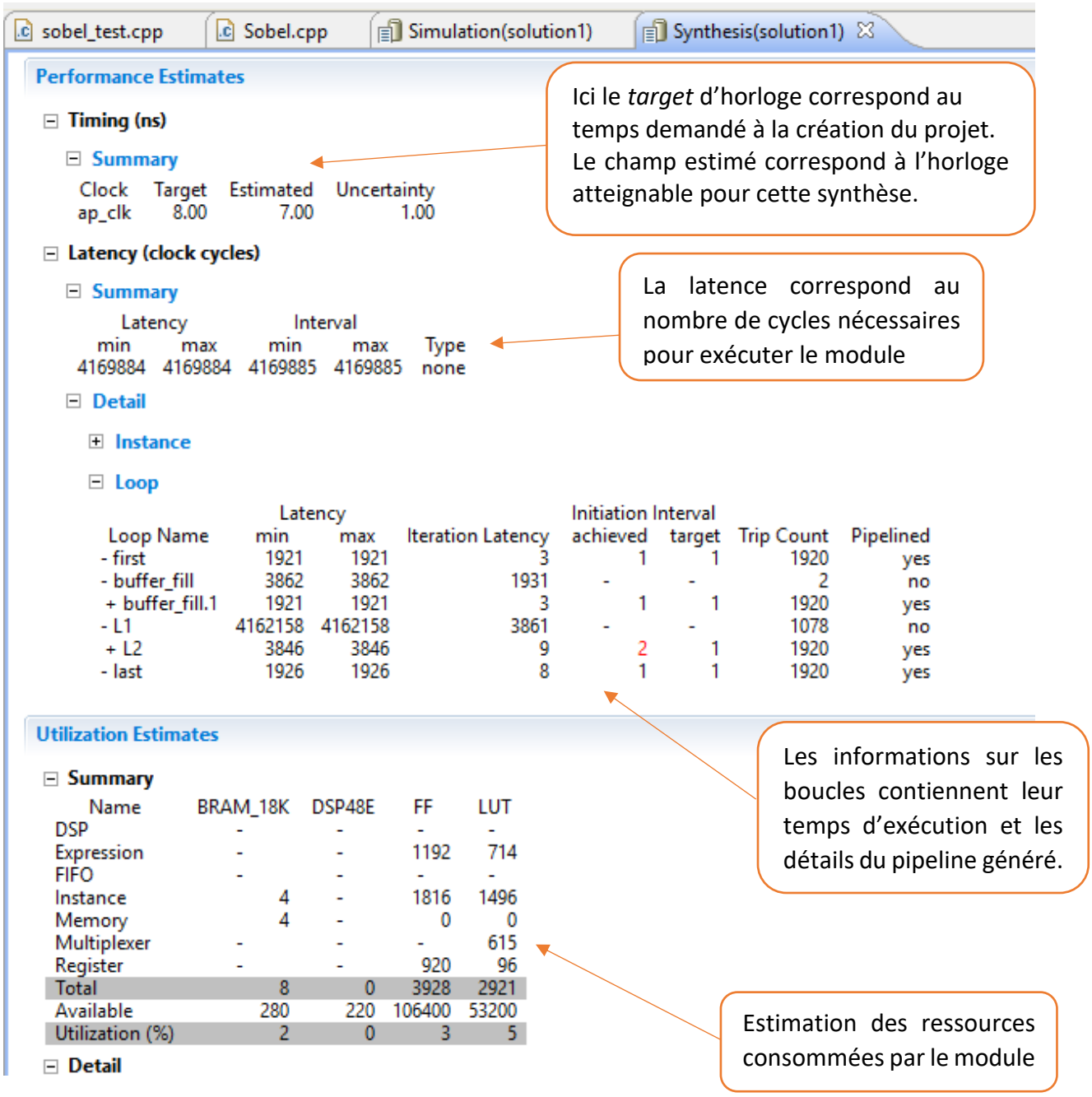
- Sans surprise, de synthétiser votre module



- 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 synthétisé[4].



- 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).

[4] 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.

II. 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 (Figure 1):

Le *pipeline HDMI* : est composé de 1) une suite de modules (IP) effectuant la transformation d'un tableau en pixels sous format RGBA (RGB + alpha, soit 4 octets/pixel) à une vitesse acceptable pour le transfert au transmetteur HDMI et 2) 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 octets[5]. 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 I^2C (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 pixels noir et blanc, donc 1 octet/pixel), et d'écrire en mémoire le résultat (sous forme de pixels 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 séparé[6]). 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.

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

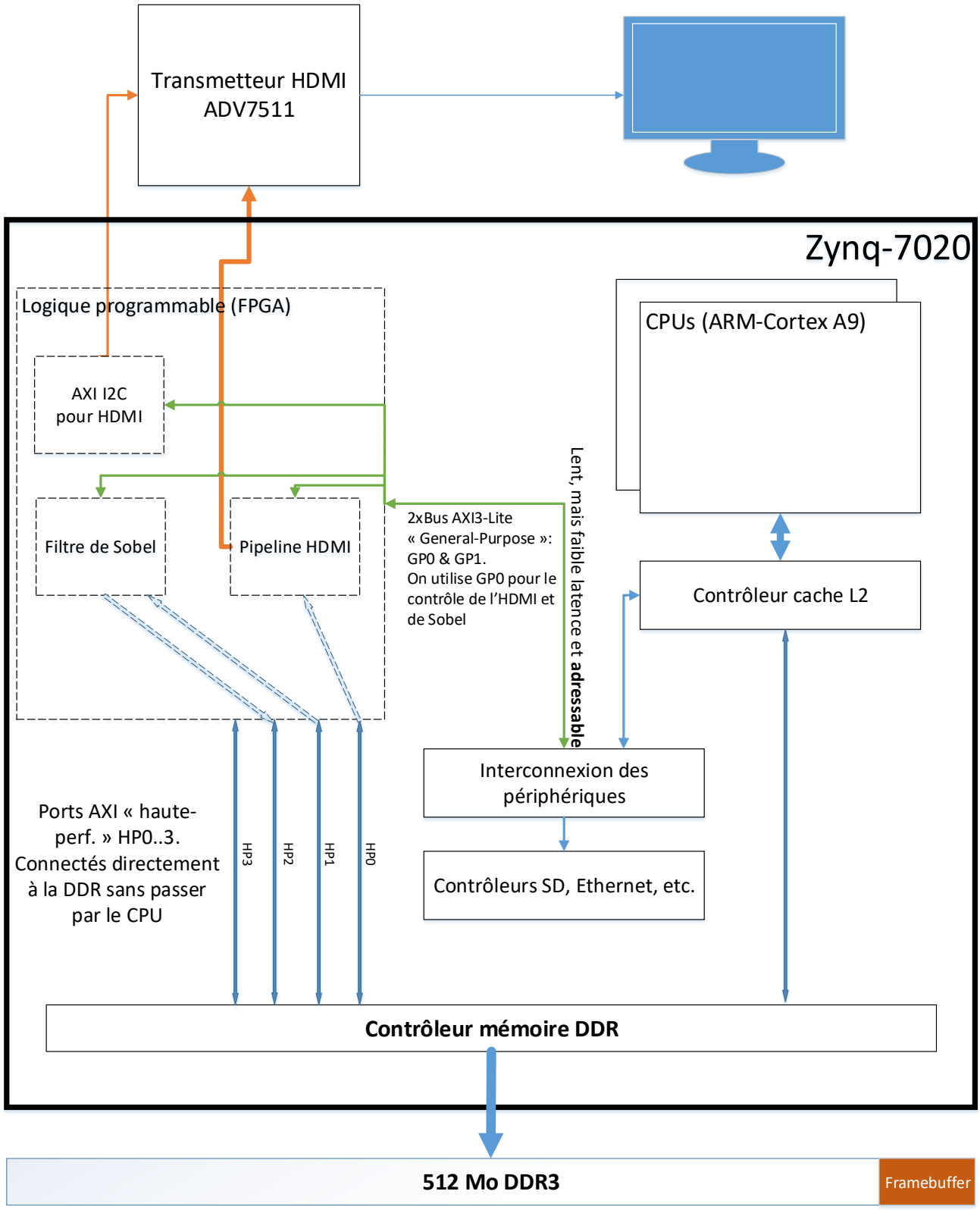


Figure 1. Diagramme simplifié des interconnexions du Zedboard (Nous verrons en classe le protocole AXI et les ports HPi)

III. Travail à effectuer

1. 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

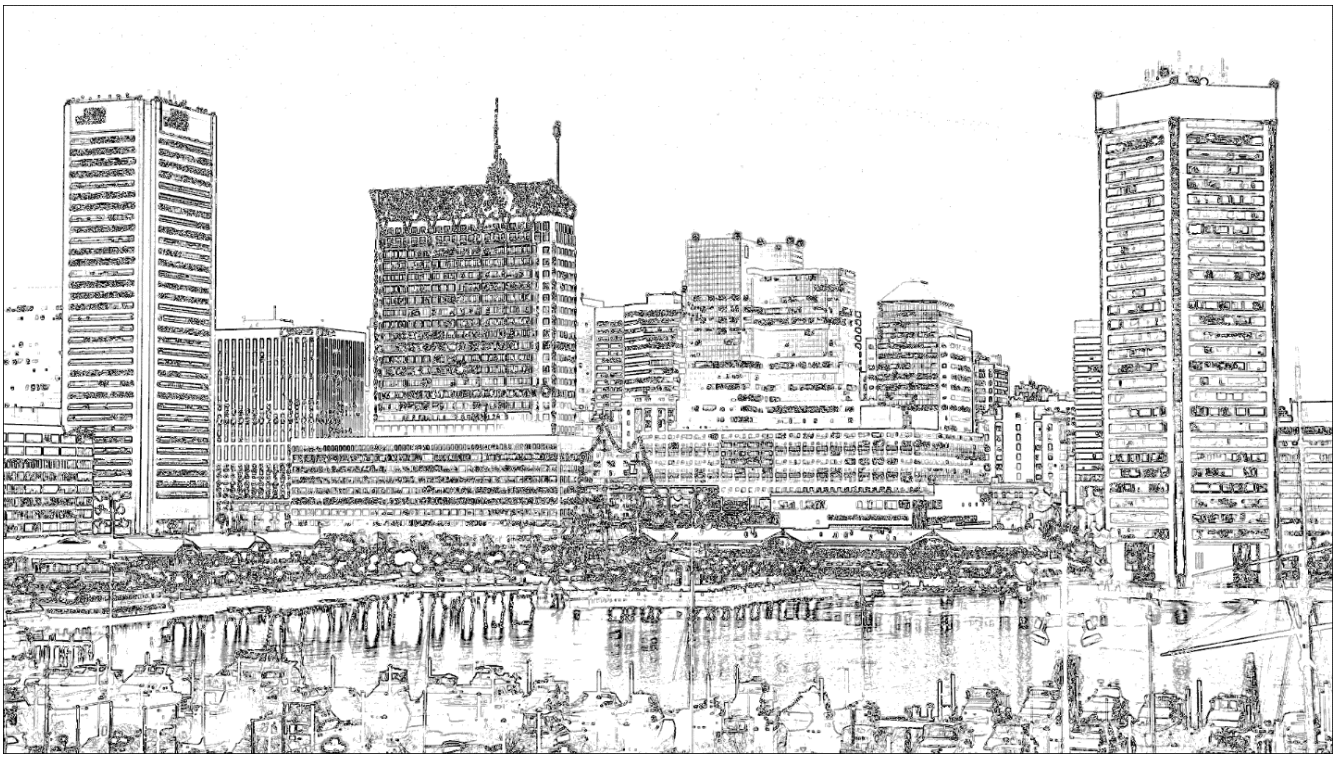


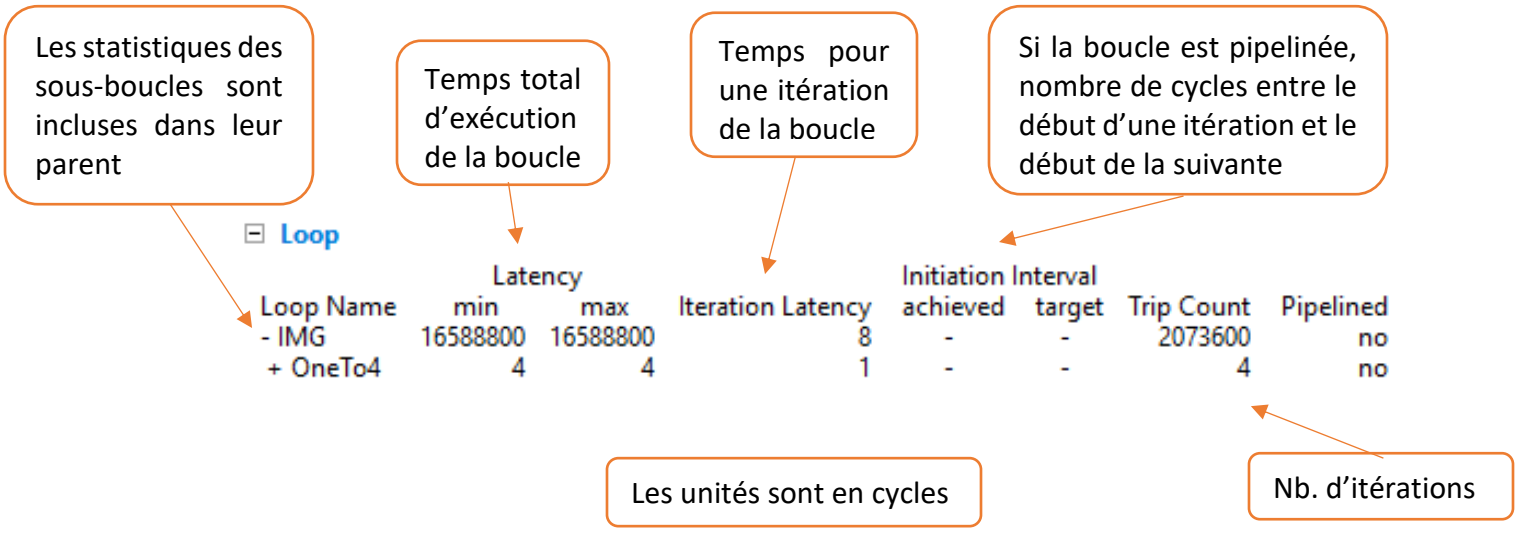
Figure 2:Résultat de l'application du filtre de Sobel sur l'image précédente

Implémentation sur Vivado HLS

Ouvrez Vivado HLS 2018.1, puis ouvrez le projet SobelLab4, préalablement copié dans *C:/TEMP/3610_4/matricule1_matricule2*.

Étapes à suivre et questions

Étape 1) 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.



Étape 2) Ajoutez, sous la boucle OneTo4, la ligne *#pragma HLS unroll*, puis resynthétisez et re-notez les mêmes métriques (comme à l'étape 1).

```
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;
    }
```

Étape 3) Enlevez le pragma, puis répétez l'étape 2 avec les pragma suivants :,

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

Importation de votre code du lab 3

Importez et modifiez 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[7], 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). **Il est fortement suggéré d'utiliser l'union OneToFourPixels fournie pour faire cela.** Si vous n'êtes pas familier avec les unions, vous pouvez consulter ce tutoriel (ou Google en général).
- 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*). **Pour vérifier le résultat, utilisez la simulation C**, et non RTL.
- 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.
- Votre filtre devrait être synthétisable, et le nombre d'itérations des boucles et son délai connu (pas de ?). S'il s'agit d'une plage de valeurs car votre boucle contient un traitement différent pour les bordures que pour le reste de l'image, prenez le pire cas.

2. 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.

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, en général, les vidéos sont 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

[7] Dans un vrai projet, il est entendu qu'un modèle de ces interfaces serait déjà disponible, ce qui n'est pas le cas ici.

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 *ffmepg*, 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.exe -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 *a9.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.exe -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.

./ffmpeg.exe -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 de la vidéo sur la mémoire de la carte

Note : Assurez-vous d'avoir complété les étapes 1 et 2 (juste avant *3. Ajout du filtre de Sobel à la plateforme matérielle*) du document *Projet Vivado avec HDMI sur Zedboard*.

On veut pouvoir copier l'extrait de vidéo convertie de la carte SD vers la mémoire de la carte. Pour ce faire, Xilinx fournit la librairie FAT32 pour systèmes embarqués *FatFs* (sous le nom *xilffs* dans la génération du BSP). 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.

Exécution logicielle

De retour dans Vivado HLS, une fois que votre code fonctionnel est en assez bon état (ex. si le code est synthétisable et la simulation C donne le bon résultat), copiez les fichiers Sobel.cpp et Sobel.h dans le dossier src de votre projet SDK et renommez la copie du SDK de Sobel.cpp en Sobel.c. Depuis le SDK, faites un clic droit sur le projet -> Refresh pour qu'il détecte l'ajout. Décommentez ensuite l'appel à sobel_filtrer() dans doSobelSW() (et l'include au début du fichier) et modifiez votre code du main pour appeler doSobelSW(). Vous n'avez plus besoin d'appeler show_video() à ce point-ci : il suffit que le résultat de la fonction sobel_filter() écrive directement dans le framebuffer de l'HDMI (inspirez-vous de la fonction show_video). **Notez la performance obtenue**. Ne vous préoccupez pas de l'exactitude de l'affichage pour cette exécution logicielle, seulement de la performance (le résultat devrait toutefois être bon en simulation et plus tard avec le filtre matériel).

Les étapes suivantes ont pour but de réaliser deux objectifs : 1) faire fonctionner le filtre sur la carte et 2) faire fonctionner le filtre sur la carte avec une performance acceptable.

Exécution matérielle du filtre sur la carte

Suivez la section « Ajout du filtre de Sobel à la plateforme matérielle » du document *Projet Vivado avec HDMI sur Zedboard* et prenez connaissance de la section « Problèmes avec la mise à jour du SDK ». Modifiez le code s'exécutant sur le Zedboard pour exécuter votre filtre matériel de manière à ce que le résultat soit affiché à l'écran (vous n'avez plus besoin de la fonction *showvideo(), ni doSobelSW()*).

Vous devez notez dans un tableau les observations suivantes:

- Le temps d'exécution réel de l'affichage d'une image.
- Le temps d'exécution du module indiqué par l'estimé HLS (nombre de cycles * la période cible, soit 10 ns à ce point).
- Les ressources consommées par votre filtre dans l'estimation HLS (BRAM, LUT, FF et DSP).

Ajoutez aussi à ce tableau vos résultats de performance logicielle obtenus plus haut.

Barème de performance du filtre matériel

La performance obtenue pour votre première itération devrait être difficilement qualifiable de fluide (en sachant que l'on cherche à filtrer et afficher la vidéo en temps réel). Vous devez maintenant améliorer cette performance, en vous aidant d'un ou plusieurs des points et conseils qui suivent. Notez que la performance obtenue correspond à 5 des 20 points de ce laboratoire, selon le barème suivant :

Images/seconde	Points (/5)
> 0	1
1+	2
2+	3
5+	4

20+	5

Notez que ce barème est interpolé linéairement entre les étapes, ex. 1,5 image/seconde donnerait 2,5 points.

Comme nous réalisons un système embarqué avec des ressources matérielles limitées, vous serez aussi pénalisé de 0.5 points si l'estimation faite par HLS de votre consommation de ressources (définie ici par la moyenne de l'utilisation de la BRAM, des FF et des LUT, en excluant les DSP) dépasse 25%. Cette pénalité augmentera linéairement de manière à ce que chaque 25% additionnel entraîne une pénalité de 0.5 points supplémentaires (ex. 24,99% = aucune pénalité, 25% = -0.5 points, 30% = -0.6 points).

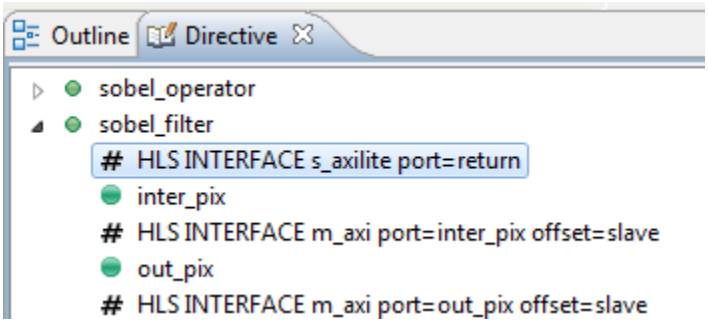
Les sections suivantes vous donnent des notions théoriques supplémentaires afin d'améliorer la performance de votre filtre matériel.

3. Amélioration de la performance du filtre matériel

Prenez conscience des points suivants pour tenter d'améliorer les performances de votre filtre. Suivez la section « Mise à jour du filtre de Sobel » du document *Projet Vivado avec HDMI sur Zedboard* lorsque vous voudrez tester des modifications sur la carte. **Pour chaque synthèse que vous ferez, notez la performance obtenue** (sous la forme des mêmes points que pour la première exécution sur la carte).

N'hésitez pas à modifier n'importe quel élément du filtre (y compris la fonction *sobel_operator()*) pour parvenir à la performance désirée.

Notez aussi l'onglet *Directive* dans le panneau d'éditeur de droite de Vivado HLS, qui vous permet d'avoir une vue sur toutes les directives (*pragmas*) que vous mettrez dans votre code. Un double-clic sur une directive vous permet de la modifier, en affichant toutes les options disponibles.



Cache personnalisée pour le module

Les données utilisées par votre filtre sont stockées dans une mémoire vive présente sur la carte à l'extérieur du FPGA. Comme avec un processeur plus traditionnel, il y a donc un délai significatif (relativement à la fréquence de l'accélérateur) entre le moment où une donnée est demandée à la mémoire et le moment où elle est reçue par le module. Il convient donc de

minimiser les accès vers cette mémoire principale et d'avoir une mémoire cache, plus proche et rapide, pour les données destinées à être réutilisées. Cependant, contrairement à un processeur plus général, nous connaissons exactement les données dont nous avons de besoin et le temps pour lequel nous en avons de besoin, ce qui nous permet de créer une structure de cache personnalisée, beaucoup plus petite et efficace qu'une cache plus traditionnelle qui stocke des informations sans savoir si elles seront encore utiles.

Vivado HLS considère tous les tableaux de variables locaux à une fonction comme une cache personnalisée, implémentée en BRAM. Ainsi, si un accès est fait par un pointeur en entrée, cet accès correspond à un accès dans la mémoire vive externe, avec une latence significative, alors qu'un accès fait dans un tableau local correspond à un accès dans une BRAM voisine, accessible en un cycle. Notons aussi que, de manière similaire, Vivado HLS considère toutes les variables locales comme des registres (implémentées avec des LUT).

Comme mentionné dans le lab 3, dans le cas d'un filtre d'une convolution 2D comme le filtre de Sobel, une architecture de cache personnalisée efficace consiste en un *line buffer*. En effet, chaque pixel traité n'a besoin que des pixels de la ligne précédente, de sa ligne et de la ligne suivante[8]. Une quatrième ligne pourrait être chargée en parallèle.

Finalement, si vous décidez de mettre en cache des pixels de votre module, notez que contrairement au lab 3, il n'est pas nécessaire de procéder par une fonction ou module externe. Vous pouvez simplement allouer un tableau local (ex. *uint8_t lineBuffer[nbCol][taille_ligne]*) et stocker et lire vos données dans ce tableau.

Transaction *burst* et flot des données

Tel que déjà vu ou vu sous peu en classe (Interconnexions sur Moodle), des bus comme AXI (utilisé par votre module pour ses entrées/sorties) possèdent la capacité de faire des transferts en rafale (ou *burst* en bon français). Comme il est montré à la figure 4[9], ceci permet de recevoir plusieurs données correspondant à des adresses successives, plutôt que de demander une donnée, attendre la réponse, demander une autre donnée, attendre la réponse, etc. comme à la figure 3. Ceci permet d'augmenter significativement le débit du transfert entre la mémoire principale et le module, une suite de transactions sans rafales étant beaucoup plus sensible à la latence entre le début de la requête et la réponse (et cette latence est particulièrement grande dans le cas de communications avec une mémoire externe).

[8] Notons aussi qu'il y aurait moyen d'être encore plus efficace, en ne conservant que 2 lignes et 2 pixels (plus un troisième en entrée à chaque itération). Cette dernière manière de procéder n'est cependant pas nécessaire, le gain étant négligeable par rapport à la complexité apportée et le temps disponible pour faire le lab.

[9] Notez que ces figures sont **très simplifiées** dans le but d'expliquer rapidement le concept. Ces figures ne prennent pas en compte les signaux de contrôle (ex. *ready* et *valid* pour AXI), le pipelinage de transactions, la réception en désordre des données, etc. Pour une vision plus exacte (et plus complexe) de la réalité, voir les notes de cours.

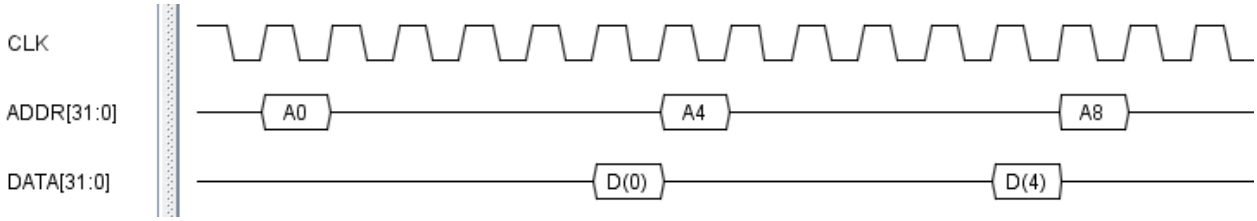


Figure 3: Transaction sans rafale (très simplifiée)

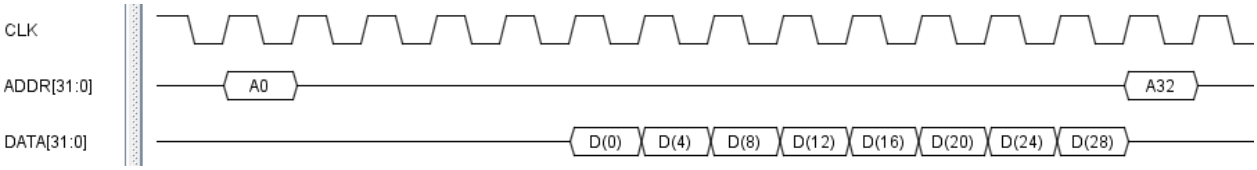


Figure 4: Transaction avec rafale (très simplifiée)

Vivado HLS génère automatiquement des transactions *burst* si les adresses des transactions sont en ordre croissants et contiguës. Vous avez donc intérêt à accéder à votre image séquentiellement, dans l'ordre de la figure 3. Vous pouvez vérifier que HLS détecte bien les conditions nécessaires à la génération de *burst* en cherchant cette mention dans le texte de sortie de la synthèse, par exemple :

```
INFO: [XFORM 203-811] Inferring bus burst write of length 1920 on port 'gmem2' (SobelLab4/Sobel.cpp:102:4).
INFO: [XFORM 203-811] Inferring bus burst read of length 1920 on port 'gmem' (SobelLab4/Sobel.cpp:103:4).
```

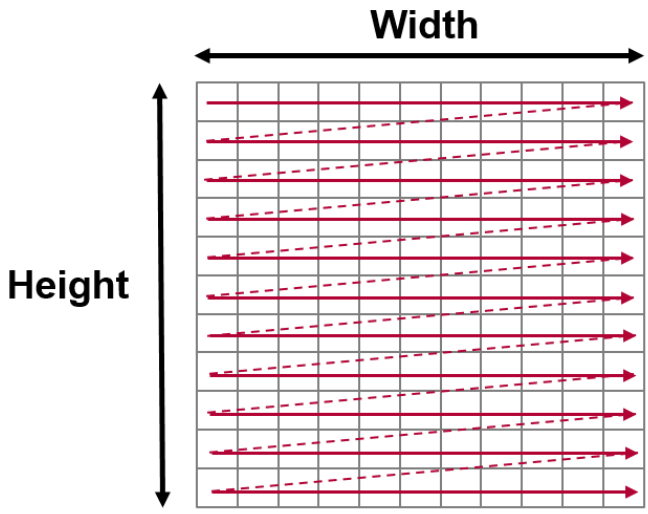


Figure 5: Ordre d'accès optimal

Déroulement de boucles

Tel que vu dans les notes sur HLS du cours et dans les questions initiales, vous pouvez dérouler des boucles pour leur exécution parallèle avec la directive *#pragma HLS unroll*. Notez que le temps de synthèse et les ressources consommées peuvent augmenter significativement avec le déroulage de boucles. Il est donc très fortement déconseillé de dérouler une boucle

parcourant une ligne entière de l'image… Le paramètre *factor* pourrait être utile dans ce dernier cas.

Pipelinage

Comme vu précédemment, il est possible de pipeliner les boucles pour augmenter leur débit avec la directive *#pragma HLS pipeline*. Il est important de noter les points suivants :

1. Vous cherchez à limiter l'intervalle d'initiation (II), qui correspond au nombre de cycles entre le début d'une itération et le début de la suivante, le plus possible. Cependant, il peut y avoir des conflits dans le partage de ports d'accès à la mémoire, etc. qui augmentent cet intervalle.
2. La directive pipeline déroule implicitement toutes les boucles à l'intérieur de la boucle pipelinée. Cependant, le déroulement de grosses boucles (ex. à 1920 itérations) causera un temps de synthèse et une consommation de ressources excessive[10]. **Pipelinez donc les boucles intérieures plutôt qu'extérieures.**
3. La directive pipeline essaie de combiner (*flatten*) la boucle sur laquelle elle est appliquée aux boucles supérieures, pour obtenir un pipeline plus long et efficace. Cependant, cette opération est faite de manière à ce que la détection de séquentialité des transactions mémoire nécessaire à la génération de *bursts* ne fonctionne plus. Il est donc nécessaire d'accompagner les directives de pipeline par la directive ***#pragma HLS loop_flatten off***.

La directive pipeline devrait donc être insérée comme ceci :

```
    for (int i = foo; i < SIZE_1; ++i) {
        for (int j = bar; j < SIZE_2; ++j) {
#pragma HLS pipeline
#pragma HLS loop_flatten off
        }
    }
```

Partitionnement de tableaux

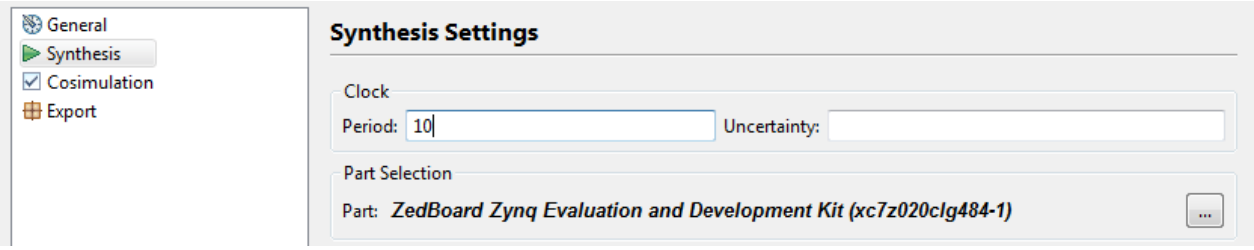
Nous avons vu plus haut qu'il est possible d'utiliser des blocs de BRAM comme cache. Cependant, bien que cela améliore la situation, chaque bloc de BRAM ne possède que 2 ports d'entrée/sortie (ils ne peuvent donc que lire/écrire 2 éléments par cycle). Or, il se peut qu'en déroulant ou en pipelinant des boucles, vous ayez à accéder à plus de deux éléments par cycle. Vous pouvez contourner ce problème en partitionnant vos tableaux dans plusieurs BRAM différentes à l'aide de la directive ARRAY_PARTITION, décrite à la page 13 de l'Introduction à Vivado HLS. Attention! Encore une fois il ne faut pas abuser de cette directive car elle risque de coûter cher en ressource.

Modification de la fréquence

Il est fort probable qu'il soit possible de synthétiser votre code à une plus grande fréquence que le 100 MHz initialement configuré pour le laboratoire (ou en d'autres mots une plus petite période

[10] C'est-à-dire tellement élevée que cela vous prendra le plus gros FPGA de la série et qui sera très dispendieux à l'achat.

que 10 ns). Pour modifier la période de l'horloge ciblée, il suffit d'aller dans le menu Solution/Solution Settings/Synthesis, puis de resynthétiser.



Notez que Vivado HLS se sert de la fréquence demandée comme guide pour savoir à quel cycle mettre chaque opération et la quantité de ressources à utiliser : il mettra plus d'opérations par cycle si la période est plus grande et moins si elle est plus petite. Ainsi, ce n'est pas parce que la fréquence estimée du design synthétisé est proche de la limite qu'il est impossible de faire mieux. Par contre, il se peut aussi que le nouveau design ait une plus petite période d'horloge, mais prenne plus de cycles pour faire son traitement (puisqu'il peut faire moins d'opérations par cycle). Ce dernier détail est cependant en très grande partie mitigé dans les designs pipelinés (où Vivado HLS ne fait que rajouter des étages au pipeline).

Une fois votre module synthétisé par Vivado HLS, vous pouvez le réexporter vers Vivado. Il faut ensuite **modifier la fréquence de l'horloge** du module (**FCLK_CLK3**) pour qu'elle corresponde à la période du design.

Conseils généraux

Même en appliquant tous les points mentionnés ici, il se peut que votre design soit sous-optimal. Portez donc attention à la sortie de la console lors de la synthèse (surtout le texte en bleu), qui peut parfois vous guider vers un meilleur résultat. Essayez aussi d'avoir un flot de données et de contrôle régulier (par exemple, traiter le cas particulier des bordures dans la même boucle que le reste de l'image). N'oubliez pas que votre code n'est plus séquentiel : il y a un parallélisme généré implicitement si vous avez plusieurs lignes de code faisant des choses indépendantes.

Questions

Analyse/Question 1a. Remettez le tableau contenant la performance de toutes vos itérations et annotez-le des modifications que vous avez fait entre chaque itération, expliquant la différence de performance.

Question 1b. Pour chacune de vos itérations, comparez la performance prévue par Vivado HLS (nombre de cycles x période d'horloge) et celle obtenue en réalité. Les deux valeurs sont-elles similaires? Assez différentes? Différentes dans certains cas et similaires pour d'autres? Expliquez ces résultats.

Des questions additionnelles vous seront fournis d'ici le 15 novembre.

Questions supplémentaires optionnelles mais appréciées.

a. Donnez une appréciation générale du lab. Par exemple, y-a-t-il des points qui manquent de clarté? Des points à améliorer? Devrait-il être redonné la session prochaine?
b. Combien de temps avez-vous passé sur ce laboratoire?

IV. Procédure de remise

Dans le but de faciliter grandement la correction, il est attendu que la remise respecte la structure suivante, dans une archive **zip** sur Moodle :

- Votre rapport en format pdf à la racine
  - Contenant le tableau à compléter, une brève analyse de celui-ci, ainsi que la réponse aux questions. N'hésitez à inclure des graphiques supplémentaires s'ils vous semblent pertinents.
- Dans un sous dossier nommé *hls* (en minuscules) :
  - Sobel.cpp et n'importe quel autre fichier modifié ou ajouté.
  - Un fichier nommé *periode.txt* contenant **uniquement** le chiffre de période cible pour laquelle HLS devrait synthétiser votre module, qui correspond aussi à la période de FCLK_CLK3 dans Vivado. Par exemple, pour une période de 10 ns, le fichier *periode.txt* devrait contenir exactement 2 caractères, soit « 1 » suivi de « 0 ». Une absence ou malformation du fichier entraînera une correction (et évaluation de la performance) avec la fréquence par défaut de 10 ns.
- Dans un dossier nommé *src* (en minuscules) :
  - Les fichiers sources que vous avez modifié ou ajouté pour exécuter le filtre sur le Zedboard (ce qui correspond normalement à *main.c*).
- Aucun fichier binaire non nécessaire supplémentaire.

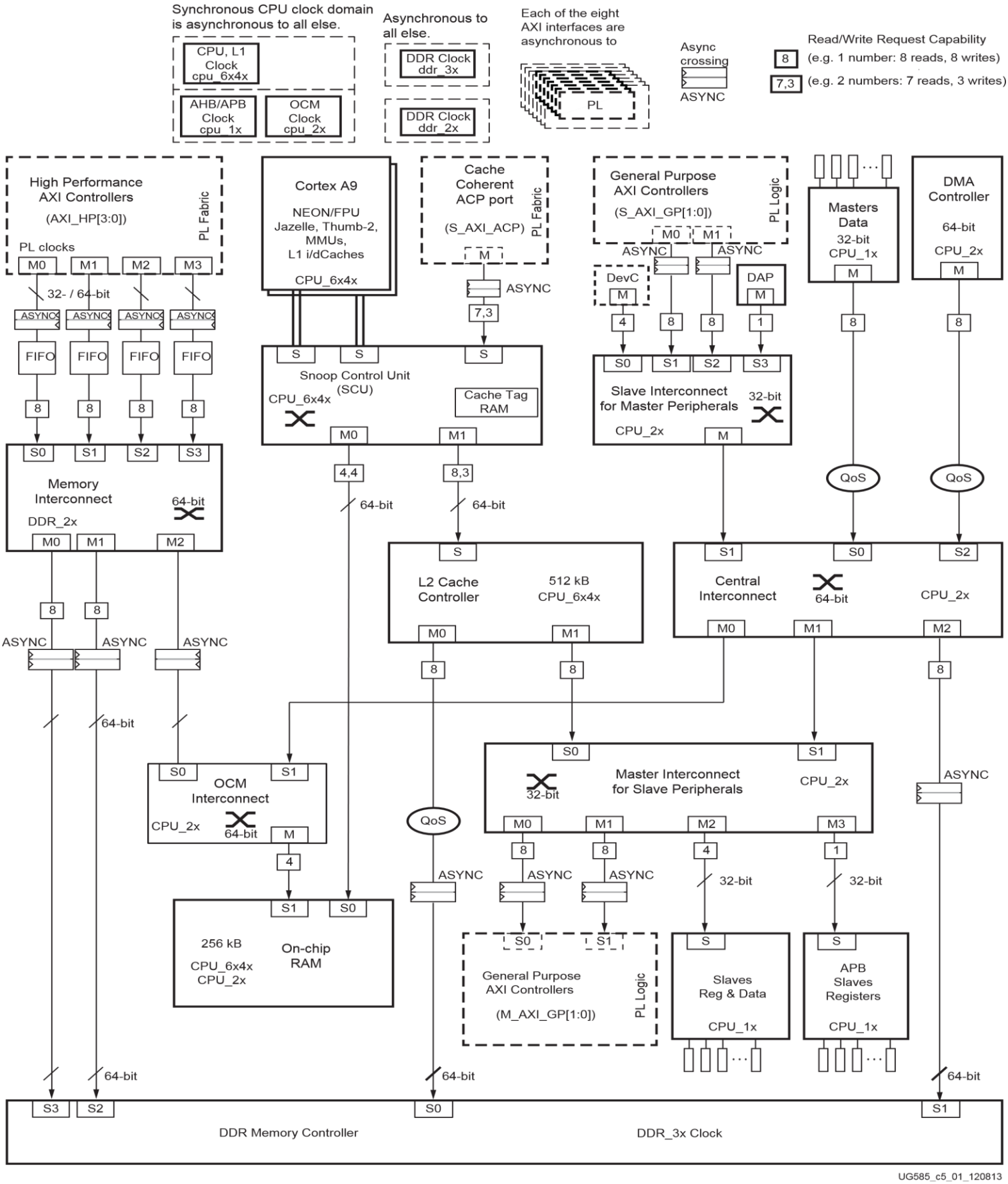
Notez qu'un non-respect de ces consignes **sera pénalisé**.

V. Barème

Barème	
Code	
Fonctionnalité (exécution correcte du banc de test et inspection visuelle de la vidéo).	/5
Performance	/5
Qualité du code	/3
Rapport	
Question 1a et b	/2
Questions à venir	/5
Respect des consignes	
Entraine des points négatifs (peut aussi invalider les points d'un exercice)	
TOTAL	/20

VI. Références

1. XAPP890 : *Zynq All Programmable SoC Sobel Filter Implementation using the Vivado HLS Tool*
2. Wikipedia: *Sobel operator*
3. Wikipedia: *YUV*
4. Tanner Helland: *Seven grayscale conversion algorithms*



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