INF3995 – Projet de conception d'un logiciel embarqué

TP2 : Sortie HDMI sur Zedboard

Jérôme Collin

(et Philippe Proulx)

Département de génie informatique et génie logiciel
Polytechnique Montréal

Sommaire

- **Objectifs** :
  - Utiliser la logique du PL du Zedboard pour compléter la logique nécessaire à procurer un traitement vidéo pour la sortie HDMI;
  - Comprendre le format Bitmap;
  - Avoir une stratégie de chargement de fichiers;
- **Date de remise** : aucune, travail d'équipe;
- **Travail préparatoire** : aucun;
- **Documents à remettre** : aucun. Par contre, le code écrit pour cet exercice et les manipulations effectuées dans Vivado sont sujets à un quiz à venir;
- **Lecture recommandée** :
  - Les notes de cours, particulièrement la présentation sur l'architecture Zynq;
  - l'article *BMP file format* de Wikipédia (décodage Bitmap).

Introduction

La carte Zedboard est suffisamment puissante pour alimenter une sortie vidéo en haute définition avec sa sortie HDMI. Une puce Analogue Devices AVD7511 est présente sur la carte pour générer les signaux de bas niveau qui s'en vont directement au connecteur et au câble HDMI. Néanmoins, il reste beaucoup de traitement à faire avant d'être capable de manipuler une image dans un programme sur un des processeurs ARM pour envoyer des informations à cette puce. Le manufacturier en est pleinement conscient et veut bien démontrer toutes les capacités de sa puce. C'est pourquoi il a placé sur Github le lien manquant entre le logiciel sur le PS et sa puce électronique en plus de fournir des explications. Il faut savoir que cette puce n'est accessible que sur le PL et qu'il faut donc construire le matériel manquant pour faire le pont.

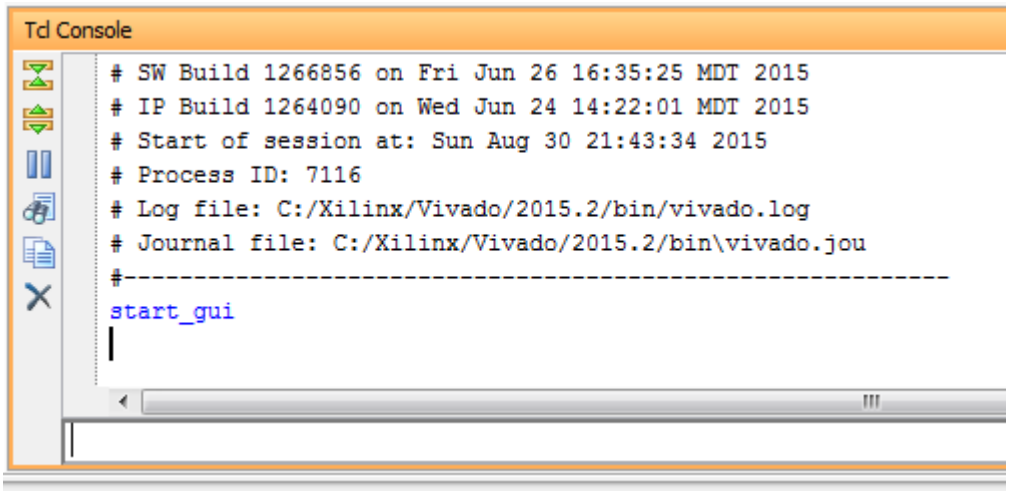
Il y a deux répertoires sur Github : le premier se nomme hdl et se subdivise en libraries et projects. Le second répertoire, no-OS, contient du code d'application pouvant s'exécuter sur le matériel réalisé à partir de ce qu'on trouve dans le répertoire *hdl*. Le sous-répertoire *libraries* contient plusieurs *IP cores*, beaucoup étant écrits en Verilog et partageant des modules communs présents dans le sous-répertoire appelé *common*. Nous utiliserons quelques-uns de ces *cores* pour ce laboratoire. Le répertoire *projects* contient des assemblages de différents *cores* pour différents projets et différentes cartes à FPGA parmi les plus courantes, dont le Zedboard. La compilation matérielle peut se faire pour les outils Quartus de Altera (acheté par Intel récemment) ou pour Vivado de Xilinx. Des scripts se chargent des étapes de compilation pour chacune des plates-formes sans qu'on doive tout refaire à la main les étapes. Il s'agit donc d'un assemblage complexe multi-puces, multi-cartes, multi-plates-formes (ouf!) particulièrement bien réalisé et clair.

On a donc tout à notre disposition pour commander directement la sortie HDMI à partir d'un code C de démonstration, en théorie. Dans la pratique, ce qui a été mis en place est pour une plus vieille version de Vivado et il y a quelques bogues à contourner pour arriver à faire fonctionner l'ensemble (prévoir un minimum de 200 heures pour y arriver pas vous-même avec

acharnement si vous souhaitez prendre cette voie, expérience vécue…) On vous propose une version réduite et mise à jour pour la version de Vivado installée au laboratoire. En gros, cette version réduite n'est que pour le Zedboard, que pour la puce ADV7511 et ne contient que les *IP cores* nécessaires à la construction du système. De plus, les répertoires ont été réorganisés pour être manipulés plus facilement. Il devrait donc être nettement plus simple de charger ce code sur la carte. Par contre, il faut prévoir plusieurs minutes pour permettre la construction du matériel et l'exécution des étapes de synthèse et d'implémentation.

Étapes de construction matérielles

Premièrement, prendre le fichier tp2Code.zip sur le site Moodle du cours (juste sous la description de ce laboratoire) et le déposer dans un endroit pour travailler. L'extraire de son format comprimé. Par la suite, démarrer Vivado directement. Par contre, ne pas créer ni ouvrir de projet! Diriger plutôt votre curseur de souris tout au bas de l'interface vers l'interpréteur de commande en langage Tcl (explication de Tcl sur Wikipédia) faisant partie intégrante de l'outil.



De là, taper la commande Unix `pwd` pour savoir dans quel répertoire l'outil travaille présentement. Normalement, il devrait s'agir du répertoire d'installation de l'outil Vivado. Ce n'est pas ce qu'on veut. Avec la commande `cd`, faire en sorte de vous retrouver dans le répertoire créé lors de la décompression du fichier tp2Code.zip.

De là, la commande `ls` devrait vous permettre de voir qu'il s'y trouve un fichier nommé `system_project.tcl`. Tapez la commande « `source system_project.tcl` », toujours dans l'interpréteur de commandes, et regardez ce qui se passe…

Normalement, si tout se passe bien, vous verrez dans le « Bloc Diagram » apparaître de nombreux blocs matériels et les interconnexions entre eux. Les étapes sont assez longues. Les étapes de synthèse et d'implémentation ne sont pas moins courtes, tout au plus beaucoup plus silencieuses…

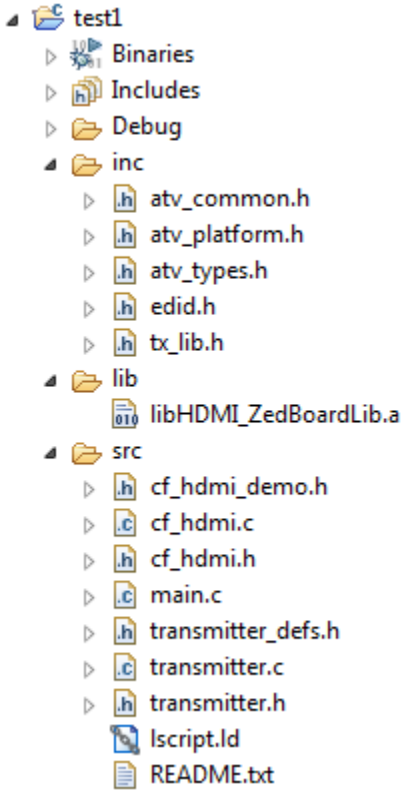
Une fois le matériel construit, prendre quelques secondes pour regarder le résultat (c'est tout de même impressionnant!) et compléter manuellement l'exportation du matériel vers SDK comme

lors du laboratoire 1 et du tutoriel (*file ->Export->Export Hardware…, Include Bitstream,* partir SDK…)

Partie logicielle

On vous laisse le soin de travailler la partie logicielle en vous suivant de ce qui est donné par le manufacturier. Le code se trouve tout de même dans le répertoire *no-OS/avd7511* sur Github (pas besoin de compte). De là, vous trouverez un sous-répertoire nommé simplement *zed* et qui contient les fichiers du code d'application (*.c et *.h). Tous sont nécessaires au bon fonctionnement. Aussi, vous trouverez un sous-répertoire *library/zed* où vous trouverez des fichiers *include* et un fichier .a qui devront être placé dans un BSP au bon endroit.

Notez que dans les étapes données par Analog Devices, le BSP est construit au même moment que l'application, et non avant l'application comme on l'a fait lors du tutoriel. Ça ne change strictement rien. C'est juste une autre façon de faire les choses. Bien essayer de reproduire la structure de fichiers qui est montrée. La commande « File->Import » peut s'avérer très utile. Notez aussi qu'une librairie en format binaire est fournie, mais il suffit de l'importer au bon endroit et elle ne sera pas recompilée évidemment. La compilation de l'application est normalement assez rapide. Une fois complétée, votre arborescence de fichiers de l'application devrait ressemblée à ceci (elle est montrée ici car ce n'est pas si évident dans la description) :



Comme le code recompile automatiquement après qu'un nouveau fichier apparaisse ou soit sauvegardé, il est bien possible que de nombreuses erreurs de compilation surviennent. Ignorez-les, le temps de tout mettre en place la structure de fichiers et que le tout soit cohérent.

Lorsque ce point est atteint, le binaire devrait être généré correctement. Après chargement du matériel et du logiciel sur la carte, il devrait être possible de voir sur le second moniteur (celui de droite) au laboratoire apparaître une image si le réglage du moniteur fait en sorte que l'entrée DVI est sélectionnée. L'image correspondante à ce qui est vu ce trouve dans le fichier `cf_hdmi_demo.h`. Son format est quelque peu cryptique, mais cette discussion nous dit que chaque valeur est en format *nnRRGGBB* où *nn* est le nombre de répétitions du pixel, et le reste est le classique rouge, vert, bleu entre 0 et 255. Il vous faudra afficher autre chose que le code de démonstration pour ce laboratoire, bien évidemment.

Tout de même, si vous voulez, vous pouvez facilement modifier le code pour faire afficher la fameuse « carte d'affaires » de Fabien Sanglard en format PPM simple. Mais, au fait, comment charge-t-on un fichier sur la carte (autrement que de « générer » d'une quelconque façon un fichier .h et de le lier à l'application comme c'est le cas ici pour la démonstration) ?

Accès à un fichier

Il y a quelques moyens pour accéder à des fichiers sur la carte Zedboard. Passer par le réseau en est un qui fera d'ailleurs l'objet d'un laboratoire séparé. Utiliser la mémoire « Serial NOR Flash Spansion S25FS256S » en est un autre, mais c'est assez compliqué. Les outils de la compagnie Spansion pour arriver à accéder au système de fichier FAT sont quelque peu obscurs…

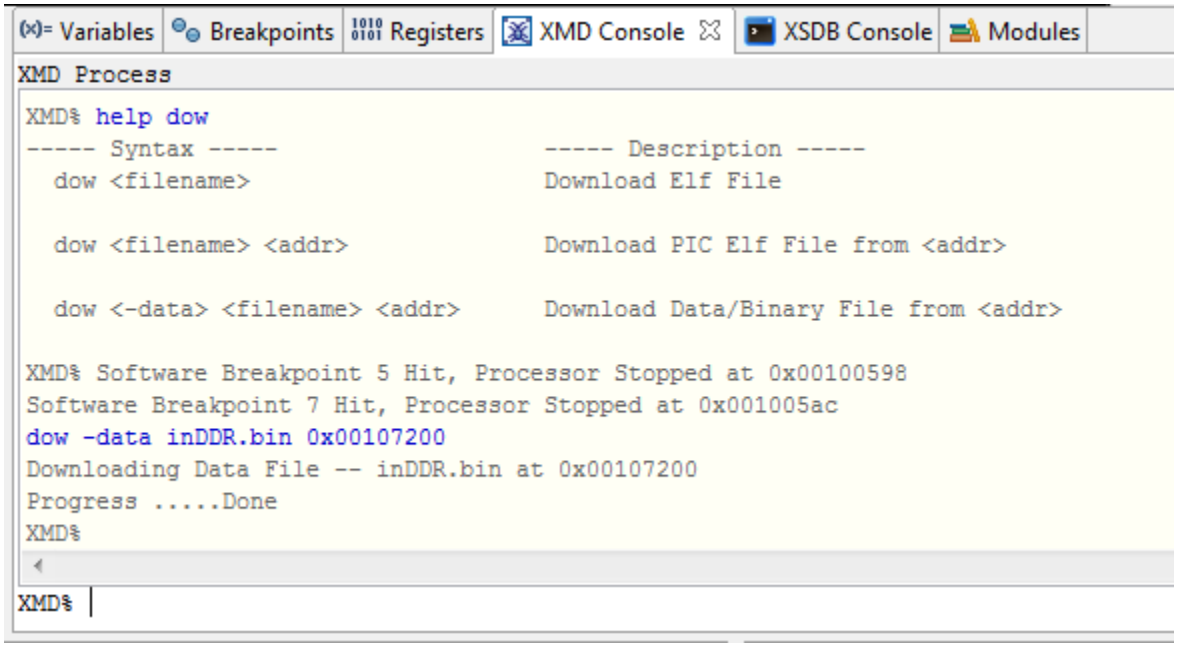
La carte SD, avec un système de fichiers FAT est beaucoup plus simple et possible. Il suffit d'utiliser le périphérique SD0 lors de la configuration du Zynq PS pour la partie matérielle et d'utiliser la librairie LibXil FFS de Xilinx pour la partie logicielle. Cette librairie peut être sélectionnée lors de la création du BSP dans SDK. Après quelques éléments de configuration, il faudra aller sur ce site FatFs - Generic FAT File System Module et surtout à la page de la description de la fonction f_open pour comprendre le fonctionnement. Par la suite, il devient possible de lire un fichier sur la carte SD en moins de 20 lignes de code. Attention : manipuler les cartes SD avec soin dans les connecteurs du Zedboard!!!

Si ceci ne vous convient toujours pas, il est aussi possible de faire ce « méga-hack » qui fonctionne plutôt bien en pratique. Créer une variable, globale ou locale, assez grosse pour contenir le fichier voulu. Dans l'exemple plus bas, la variable `buf` peut prendre facilement un fichier de 15 kilo-octets et il reste encore de l'espace. Il faut travailler avec le débogueur et s'organiser pour faire une exécution pas-à-pas. Dans les lignes qui suivent la déclaration de la variable, faire afficher son adresse et regarder le résultat au terminal. Avant d'aller plus loin dans l'exécution du programme, aller à la console XMD et télécharger en mémoire avec la commande « dow » (3 premières lettres de « download ») un fichier de données exactement à l'adresse de la variable (ici 0x00107200). C'est une injection intraveineuse directe dans le programme, disons… Si vous poursuivez plus loin l'exécution du programme avec le débogueur, la variable (ici `buf`) contiendra le contenu du fichier inDDR.bin. Ce n'est pas élégant, mais ça marche… Le problème est qu'il faut refaire l'opération à chaque exécution du programme ce qui peut être fastidieux à la longue. Donc, la carte SD pourrait s'avérer un bon

choix à la longue pour faire les choses proprement et de façon plus efficace. Tout de même, c'est un bon truc à savoir dans certaines circonstances.

```
int main()
{
    char buf[20000];
    char *ptr = buf;

    init_platform();
    xil_printf ("Adresse de la variable sur la pile %x\n\r", (int)ptr);
...
```



Travail logiciel demandé

Il devient rapidement difficile de manipuler des fichiers d'images qui ne sont pas dans des formats standards et qui ne sont pas supporter par les outils courants de visualisation. On vous demande donc de supporter l'affichage d'images en format Bitmap. Le véritable exercice du travail pratique, outre vous familiariser avec tout ce qui a été vu, commence donc ici.

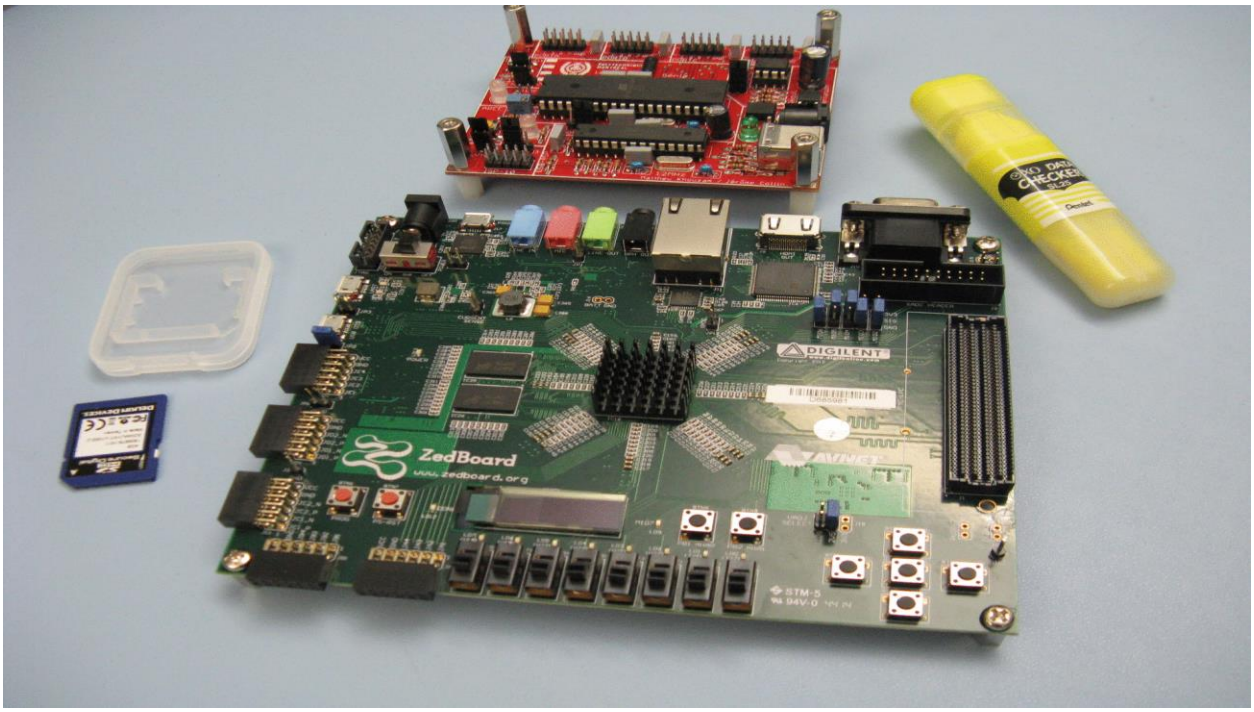


Image à afficher sur le moniteur

Le fichier Bitmap en soi est en mode de couleurs indexées : on retrouve d'abord dans le fichier une table qui associe un index à une « couleur réelle » (sur 24 bits), puis les pixels qui sont listés par la suite font référence à l'un des 256 index plutôt qu'à une couleur. Ceci permet de sauver beaucoup d'espace, mais restreint le nombre de couleurs à 256, ce pourquoi l'image semble un peu étrange. Une copie de l'image est disponible sur Moodle.

Construisez votre décodeur Bitmap dans des fichiers `bitmap.h` et `bitmap.c`, puisque vous réutiliserez ces routines au cours du projet de session. Assurez-vous de produire du code modulaire, c'est-à-dire de ne rien inclure de spécifique au TP2 dans ces fichiers. Fiez-vous à l'article *BMP file format* sur Wikipédia pour connaître le format à analyser. Ne soyez pas gêné d'ouvrir l'image fournie avec un éditeur hexadécimal et de lire le fichier en suivant la description de son format pour bien le comprendre. L'outil ImageMagic est aussi bien utile parfois.

Le format BMP a été prévu pour Windows, qui roule sur des machines en *little-endian*. Le ARM, lui aussi, semble opérer en *little-endian*. En d'autres mots, vous n'aurez probablement pas à inverser les octets de certains champs de 16 ou 32 bits de l'en-tête Bitmap pour obtenir de bonnes valeurs mais il est toujours bon de savoir que la manipulation de fichiers binaires peut entraîner des erreurs dans l'interprétation de l'ordre des octets.

Remise

Il n'y a toujours pas de remise pour ce travail pratique. Tout de même, pour vous habituer, continuer de suivre les recommandations du travail pratique 1 et « nettoyer » vos répertoires pour éviter de traîner des fichiers inutiles. La remise du travail pratique 3 l'exigera (sous peine

de perte de points). Il s'agit aussi de bien partager les résultats de l'équipe à l'ensemble de ses membres. Ce travaille pratique servira également de base au projet final.