Ensimag 3A, filière SLE

2017-2018

Modélisation TLM en SystemC
TP n°3 : Intégration du logiciel embarqué

Consignes importantes pour tous les TPs

Rappel : le fichier `TP-commun.pdf` contient un ensemble de **consignes très importantes** pour les 3 TPs. Respectez ces consignes **scrupuleusement** pour ne pas perdre de points bêtement.

1 Objectifs

Ce TP s'intéresse à l'intégration du logiciel embarqué. La plate-forme à modéliser a été développée sur FPGA (cf section 9 pour un descriptif du système). C'est un petit système sur puce, avec le strict minimum pour faire tourner du logiciel non-trivial avec affichage graphique (mais pas d'accélérateur matériel ou de bloc IP exotique). Le logiciel embarqué proposé est le jeu de la vie (cf. `http://en.wikipedia.org/wiki/Conway%27s_Game_of_Life` pour les règles du jeu et des exemples rigolos).

Nous allons expérimenter deux approches en simulation :

La simulation native : le logiciel embarqué sera compilé avec le même compilateur que la plate-forme, et liée comme un morceau de code en C quelconque. Les accès mémoires pertinents du point du vue des composants matériels seront routés sur le bus TLM. Le code embarqué est encapsulé dans un composant TLM appelé « wrapper natif ».

La simulation via ISS : L'ISS, ou Instruction Set Simulator, va interpréter directement le code compilé pour le processeur cible (un Microblaze dans notre cas). On utilisera donc la même chaîne de compilation que pour l'intégration du logiciel sur la puce finale (en théorie, le même binaire, au bit près, peut tourner sur ISS et sur la puce). Toutes les lectures/écritures faites par l'ISS seront routées sur le bus.

Il n'est pas garanti qu'un logiciel développé sur ISS continuera à tourner en simulation native. Par contre, un logiciel bien écrit et qui marche en simulation native devrait marcher sans modification sur la puce ou en simulation avec ISS (après recompilation bien entendu).

Dans les deux cas, la plateforme sera « loosely timed », c'est à dire que nous nous servons du temps pour faire une simulation raisonnable (par exemple, les timers sont timés correctement, l'ISS est modélisé avec une période qui correspond à la version FGPA), mais nous ne cherchons pas la précision. Par exemple, en simulation native, nous ignorons totalement la notion de temps pour l'exécution du logiciel, le modèle de bus que nous utilisons n'est pas timé, ...

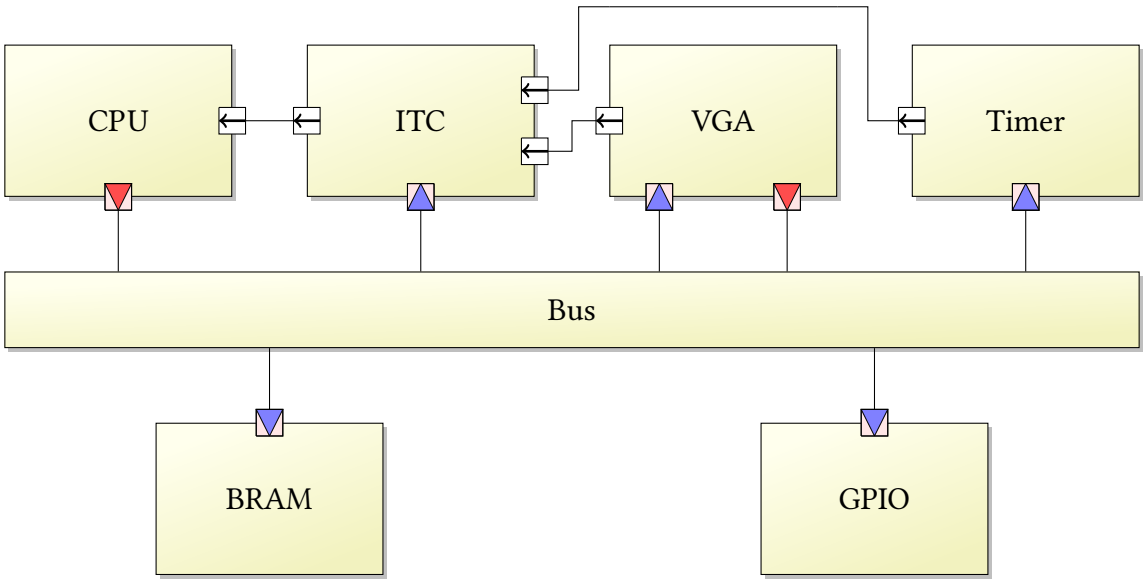


FIGURE 1 – Vue graphique de la plateforme TLM

2 Travail attendu et organisation

Pour mémoire, ce TP est, comme le précédent, noté, et doit être fait en *binôme*, mais comme vous êtes 27, il faudra que l'un d'entre vous soit seul ! Il compte pour $\frac{2}{3}$ des 50% de la note finale, les 50% autres venant d'un examen papier, calqué sur les précédents. Le rendu attendu est un fichier texte avec l'url d'un git contenant votre rendu. Attention, le dernier commit pris en compte sera celui de la deadline attendue, le lundi 22 janvier 2017 à 8h00, heure de Grenoble.

Le déroulement des séances que nous vous suggérons est le suivant :

— 1er TP : simulation native fonctionnelle ;
— 2nd TP : simulation à l'aide de l'ISS sans les interruptions ;
— 3ème TP : simulation à l'aide de l'ISS avec les interruptions et implantation sur la carte FPGA.

Si vous dérivez fortement, nous vous conseillons de travailler entre les séances.

3 La plateforme

La plateforme modélisée est celle présentée dans le document `TP3-zybo.pdf`. Les détails non-pertinents au niveau TLM sont abstraits. La plate-forme obtenue est décrite dans la figure 1.

4 Point de départ

Pour ce TP, on ne réutilisera pas les composants des TPs précédents. Récupérer le squelette de plate-forme dans `TPs/squelette/tp3` dans votre archive Git.

Le répertoire est organisé en plusieurs sous-répertoires :

zybo/ la « vraie » plateforme, prévue pour tourner sur carte FPGA Zybo, connectée à un écran VGA.

hardware/ les composants HW de la plateforme.

native-wrapper/ le toplevel (i.e. le fichier `sc_main_XXX.cpp` contenant la fonction `sc_main`) et les composants spécifiques au wrapper natif.

iss/ toplevel et composants spécifiques à l'ISS microblaze.

elf-loader/ le chargeur de fichier elf et ses dépendances. Utilisé par Memory.cpp pour charger le logiciel embarqué au format ELF lorsqu'on fait une simulation avec ISS.

software/ Le logiciel embarqué. Pour nous, il sera très simple : c'est une implémentation du jeu de la vie, qui tient en un seul fichier (`main.c`).

software/cross/ le nécessaire pour la compilation croisée (cross-compilation) du logiciel embarqué.

software/native/ le nécessaire pour la compilation en wrapper natif.

Pour commencer, on peut essayer :

```
cd native-wrapper
make -k
cd ../iss/
make -k
```

(l'option `-k` de `make` permet de ne pas arrêter la compilation à la première erreur rencontrée)

Le premier `make` va échouer car vous devez définir les règles de compilation pour la version native du logiciel embarqué. Le fichier à modifier est `squelette/tp3/software/native/Makefile`. Le second `make` va échouer sur la compilation du logiciel embarqué, mais construit tout de même l'exécutable `run.x` pour la plateforme. Le fichier à modifier se trouve dans `squelette/tp3/software/cross/Makefile`. Une fois le TP terminé, les mêmes commandes créeront `native-wrapper/run.x` et `iss/run.x`, correspondant aux deux versions de la plate-forme.

La suite du TP sera donc de reconstituer les différents points qui manquent pour la compilation et l'exécution de ces deux plate-formes.

5 Compilation pour simulation native

```
cd ../
cd software/native/
cat hal.h
make
```

Peu de choses à faire dans ce répertoire : il manque simplement la règle pour compiler le logiciel embarqué en mode natif. On compilera avec un compilateur C (`gcc`) et non C++ (`g++`).

Le fichier `hal.h` est complet, mais il se contente de rediriger les appels sur des fonctions qui seront implantées dans le wrapper natif (`native-wrapper/native_wrapper.cpp`).

6 Execution en simulation native

```
cd ../../
cd native-wrapper/
make
```

Pour la simulation native, il reste à implémenter le wrapper natif. Il se trouve dans le fichier `native_wrapper.cpp`. Le squelette est fourni, mais le corps de la plupart des fonctions n'est pas implémenté.

- Les fonctions `hal_read32`, `hal_write32`, `hal_cpu_relax` et `hal_wait_for_irq` doivent rediriger sur les méthodes correspondantes de `NativeWrapper`.
- Les méthodes de `NativeWrapper` doivent être implémentées.

Par ailleurs, en exécution native, on peut avoir des problèmes liés au fait que par défaut, SystemC ne laisse pas le temps s'écouler. Une boucle d'attente active (polling) dans le logiciel embarqué va donc figer la simulation (et il y en a une dans `main.c` !). L'astuce classique consiste à « casser » les boucles d'attente avec un appel à `hal_cpu_relax()`, qui laisse le temps s'écouler [1].

7 Compilation croisée du logiciel embarqué

```
cd software/cross/
ls
make
```

Pour l'instant, le `Makefile` fourni n'est pas complet : il manque les règles pour compiler, assembler et lier le logiciel embarqué (qui sera le fichier `a.out`). On compilera le logiciel avec un compilateur C, des macros sont fournies en tête du `Makefile` pour les noms des commandes. Pour l'édition de liens, on utilise un « linker script », qui donne à l'éditeur de liens les adresses finales des différentes sections et de certains symboles. Le script est dans le fichier `software/cross/ldscript`, et l'option `-T` de `ld` sera nécessaire. Une fois ces règles implémentées dans le `Makefile`, les règles `make dump.dis` et `make sections.txt` fournissent un résumé « lisible » du fichier compilé.

8 Simulation avec ISS

```
cd ../../
cd iss
make
./run.x
```

La plateforme devrait maintenant compiler et être capable de charger le logiciel embarqué. Parmi les choses qui ont été faites pour vous :

- le composant `Memory` expose directement son tableau de stockage (champ `storage`) ;
- la fonction `sc_main()` charge directement le logiciel embarqué dans ce tableau, en utilisant le chargeur ELF ;
- un ISS microblaze fait partie de la plateforme, il est écrit en C++ (sans SystemC, ni TLM) ;
- un wrapper pour cet ISS (un composant SystemC, avec interface `ensitlm` in un port **`sc_in`** pour les IRQ, qui fait appel aux méthodes C++ de l'ISS) est partiellement implémenté.

Il manque cependant plusieurs choses, que nous allons implémenter dans les sections suivantes.

8.1 Gestion de la mémoire

- la couche d'abstraction (`software/cross/hal.h`) n'est pas implémentée. Tous les accès mémoires via cette API stopperont la simulation brutalement sur un `abort()` (dans un premier temps, vous pouvez ignorer la fonction `printf()`) ;
- l'ISS est totalement implémenté, mais le wrapper pour l'ISS ne l'est pas. Lorsque l'ISS demande une lecture ou une écriture, c'est au wrapper de faire l'accès effectif au bus. Il y a 3 types d'accès à implémenter : le fetch, les « load » et les « store ». Ils sont identifiés comme tels dans `mb_wrapper.cpp`. Attention, l'ISS gère en interne les données en big endian, et

1. sur la plupart des plate-formes, cette fonction `hal_cpu_relax()` aurait des choses intéressantes à faire en dehors du contexte de la simulation native, comme diminuer la priorité du processus courant, vider des caches, ... mais ce n'est pas le cas sur notre microblaze sans OS, sans cache, ...

la plateforme d'exécution est une machine Intel en little-endian, donc le reste de la plateforme s'attend à recevoir des entiers en little-endian. Les macros `uint32_machine_to_be` et `uint32_be_to_machine` peuvent aider.
— la pile est positionnée, mais pas à une adresse raisonnable. Il faut positionner correctement la valeur de `_stack_top` dans `software/cross/ldscript`.

8.2 Affichage avec `printf`

Notre plate-forme a une capacité de debug limitée (on peut activer des traces via des macros au début des fichiers `mb_wrapper.cpp` et `microblaze.cpp`, mais elles sont en général soit trop soit pas assez verbeuses ...). Pour travailler plus confortablement, il est souhaitable de pouvoir utiliser la fonction `printf` pour afficher du texte sur la sortie standard du programme SystemC. Ce n'est pas aussi simple qu'on aurait pu le croire, vu que le code exécutable est interprété par l'ISS, on ne peut pas appeler directement la fonction `printf` de notre libc (hôte) depuis le code embarqué. La solution retenue est d'avoir un composant `UART`, qui va recevoir des caractères depuis le bus `ensitlm`, et les afficher via `cout` en C++ (le composant physique aurait envoyé les caractères sur un lien série). Le composant UART vous est fourni, il vous reste :

— à instancier et connecter correctement le composant au bus dans `sc_main`.
— à écrire le corps de la fonction `printf` dans `hal.h`. On ne s'intéresse qu'au cas de `printf` à un seul argument, et sans caractères spéciaux (i.e. on affiche la chaîne passée en argument sans traitement particulier). Il suffit d'afficher les caractères un par un jusqu'au caractère `'\0'`.
— l'exécution de la fonction ci-dessus va probablement faire un accès en lecture sur un seul caractère sur le bus par l'intermédiaire de `READ_BYTE` dans `mb_wrapper.cpp`. Pour mémoire, le bus `ensitlm` ne supporte que des transferts de mots, il faut donc trouver la « bonne » adresse mot correspondant à l'adresse octet, puis extraire le bon octet du mot.
Par exemple, pour lire un caractère à l'adresse `0x016af`, il faut faire un accès à l'adresse `0x016ac` (i.e. le multiple de 4 immédiatement inférieur), qui va donner 4 octets, puis extraire l'octet correspondant via un décalage de 3 octets et un masque de bit :
`((*(uint32_t *)0x016ac) >> (3 * 8)) \& 0xF`.

Vérifier que les instructions `printf` présente dans `main.c` sont bien prises en compte (l'exécution peut être lente, mais un message doit être affiché au début de la fonction `main`).

8.3 Gestion des interruptions

La gestion des interruptions est totalement absente du wrapper. Il faudra implémenter un processus SystemC sensible aux fronts sur le port `irq`, et qui utilise la fonction `m_iss.setIrq(`**`true`**`)` pour signaler à l'ISS qu'une interruption a été reçue. Une fois l'interruption traitée par l'ISS, il faut appeler `m_iss.setIrq(`**`false`**`)`. Pour que l'ISS ait vu l'interruption, il faut que la fonction `m_iss.step()` ait été appelée plusieurs (par exemple, 5) fois. Il faudra donc ajouter un compteur qui remet l'interruption à faux après 5 cycles.

9 Passage sur carte Zybo

Pour démontrer que le logiciel peut s'exécuter maintenant sur du vrai hardware, on se propose d'implanter l'équivalent du système TLM sur FPGA.

Les sections suivantes décrivent la plateforme matérielle sur FPGA, et la procédure pour la faire tourner en pratique. La plateforme est un système minimaliste classique (processeur, mémoire, contrôleur d'interruption, timer), qui contient aussi un contrôleur VGA qui permet un affichage à l'écran.

9.1 Lancement de Vivado

Dans un terminal, lancez :

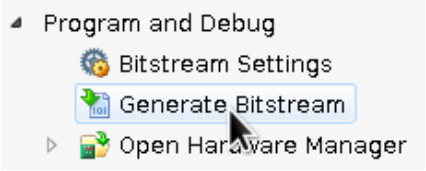
```
source /matieres/5MMMTSP/Xilinx/Vivado/2016.2/settings64.sh
vivado TPs/squelette/tp3/zybo/project_1/project_1.xpr
```

Attention, la première ligne positionne des variables d'environnement de manière très intrusive. Il est probable que la plupart des commandes autres que `vivado` cessent de marcher, mais pas de panique : la modification ne s'applique qu'au terminal courant, il suffit d'en ouvrir un deuxième en cas de problème.

9.2 Ouverture du projet

Au lancement de Vivado, faites « open project », puis choisissez le fichier `TPs/squelette/tp3/zybo/project_1/project_1.xpr`. Le projet s'ouvre et Vivado affiche une fenêtre « Project Summary » peu intéressante pour l'instant. La barre latérale de gauche affiche les différentes étapes de la conception à l'implémentation. Cliquez sur « Open Block Design » dans la rubrique « IP Integrator ». Vous devriez obtenir le schéma de la figure 2.

La synthèse étant assez longue (environ 10 minutes), lancez-la dès maintenant en cliquant sur « generate bitstream » en bas à gauche de l'écran (les étapes se font de haut en bas, mais cliquer directement sur la dernière étape enchaîne toutes les autres automatiquement : synthèse, implémentation et génération du fichier .bit) :



En attendant que la synthèse soit terminée, quelques informations sur la plateforme dans la section suivante.

9.3 La plateforme

Le composant central de la plateforme est « AXI Interconnect » : c'est le bus de notre système. On voit beaucoup de détails absents d'une plateforme TLM typique : les signaux d'initialisation (`reset`), nécessaires pour initialiser la plateforme correctement, et horloges (`clk`) qui sont évidemment nécessaires physiquement, mais également pertinentes dans une représentation logique car il peut y en avoir plusieurs : il faut préciser quel composant est connecté à quelle horloge.

Les autres composants sont :

MicroBlaze, MicroBlaze Debug Module, Processor System Reset : Le processeur de la plateforme. Le processeur MicroBlaze a la particularité d'être conçu pour FPGA (on parle de « softcore »), et d'être très simple. Le « Debug Module » permet d'accéder au processeur via le JTAG pour pouvoir utiliser un debugger, et le « Processor System Reset » génère les signaux d'horloge et d'initialisation.

AXI Interrupt Controller est un contrôleur d'interruption. Il prend en entrée un signal 3 bits. On utilise donc le module « Concat » pour passer de 3 signaux 1 bit à un signal 3 bits (ce module est nécessaire du point de vue typage, mais ne correspond à rien physiquement).

vga_axi_ip_v1_0 Le contrôleur VGA qui lit une image (1 bit par pixel) en RAM et l'affiche à l'écran.

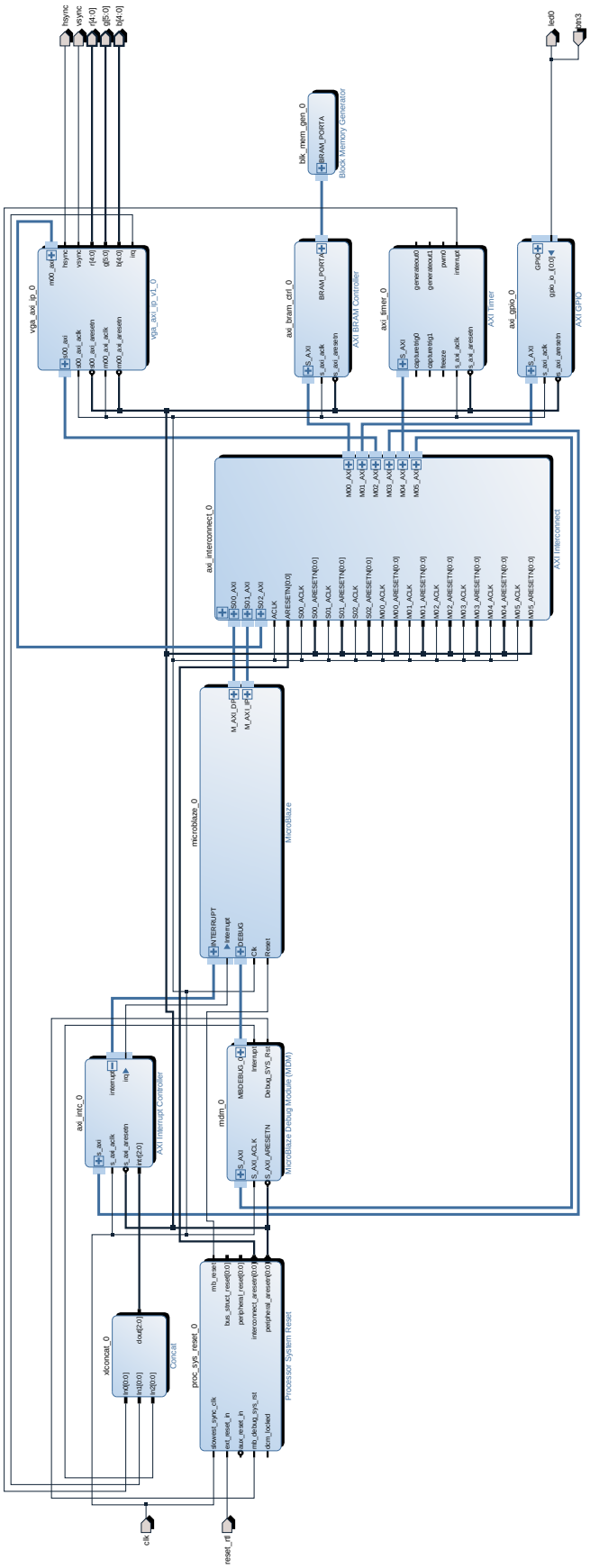


FIGURE 2 – Block Diagram de la plateforme

AXI BRAM Controller et Block Memory Generator correspondent à une mémoire. La BRAM est la mémoire embarquée dans le FPGA. Le premier module fait l'interface avec le bus AXI, et le second correspond à la mémoire à proprement parler.

AXI Timer est un timer. On le programmera pour envoyer des interruptions périodiquement pour rythmer l'exécution de la plateforme.

AXI GPIO (General Purpose Input Output) est un module d'entrée sortie. On l'utilisera pour gérer un bouton poussoir. Pour avoir un retour visuel, le bouton poussoir est également connecté à une LED.

Pour avoir les détails sur un composant, double-cliquez sur l'un d'eux. Il y a un bouton « documentation » en haut à gauche qui vous permet d'accéder au manuel et à la documentation en ligne pour chaque composant.

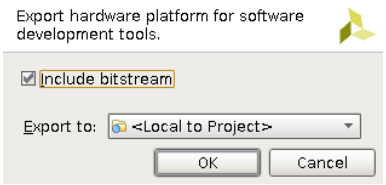
9.4 Branchement du FPGA

Même si la synthèse n'est pas terminée, vous pouvez brancher votre carte Zybo : le cordon USB sur le port USB du PC (utilisé pour programme le FPGA et pour le debug), et l'écran externe branché sur le port VGA du Zybo. Vérifiez que l'interrupteur SW4 à côté du branchement USB est bien sur « On ».

9.5 Lancement de l'environnement logiciel (SDK)

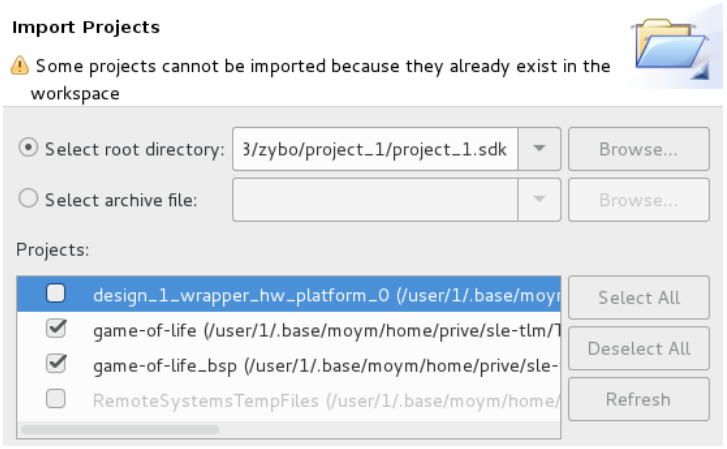
Pour cette étape, il est nécessaire que la synthèse soit terminée. Si ce n'est pas le cas, avancez sur la plateforme TLM en attendant. Quand la synthèse se termine, une fenêtre « Bitstream Generation Completed » s'ouvre : elle n'est pas importante pour nous, vous pouvez la fermer en cliquant sur « Cancel ».

Le SDK Xilinx permet de développer du logiciel embarqué paramétré par le système matériel, et permet de programmer le FPGA. Il a donc besoin d'une description du matériel, que nos allons lui fournir : depuis Vivado, faites « File » → « Export » → « Export hardware ». Cochez la case « include bitstream » et validez :



Lancez maintenant le SDK : menu « File » → « Launch SDK ». Gardez les paramètres par défaut et validez : une version spécialisée Xilinx d'Eclipse s'ouvre. L'espace de travail (*workspace*) contient pour l'instant un projet « `design_1_wrapper_hw_platform_0` » : ce projet correspond à la description du matériel que nous venons d'exporter. Le SDK l'utilise, mais nous n'y toucherons pas.

Ouvrez le projet « game-of-life » : menu « File » → « Import... ». Choisissez « General » → « Existing projects into workspace ». Choisissez « `project_1.sdk` » comme répertoire racine et importez « game-of-life » et « game-of-life_bsp » :



Le projet « game-of-life » est notre logiciel embarqué. Si vous ne connaissez pas le jeu de la vie, voir par exemple : `http://en.wikipedia.org/wiki/Conway%27s_Game_of_Life`. Le projet « game-of-life_bsp » (BSP = Board Support Package) contient le logiciel de bas niveau spécifique à notre plateforme. Il a été généré automatiquement par Vivado. Nous n'utiliserons que très peu ce BSP : le logiciel tourne sur machine nue et accède directement aux registres des périphériques. L'intérêt pour nous est de maîtriser et comprendre totalement ce qu'il se passe, mais dans la vraie vie le BSP est bien utile !

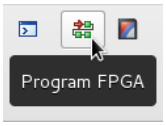
Dans le projet « game-of-life », cherchez le fichier `main.c`. C'est exactement le même que celui de la plateforme TLM.

On trouve aussi le fichier `hal.h`, qui implémente une partie des primitives vues en cours en utilisant le BSP Xilinx (`Xil_In32`, `Xil_Out32`, `xil_printf`).

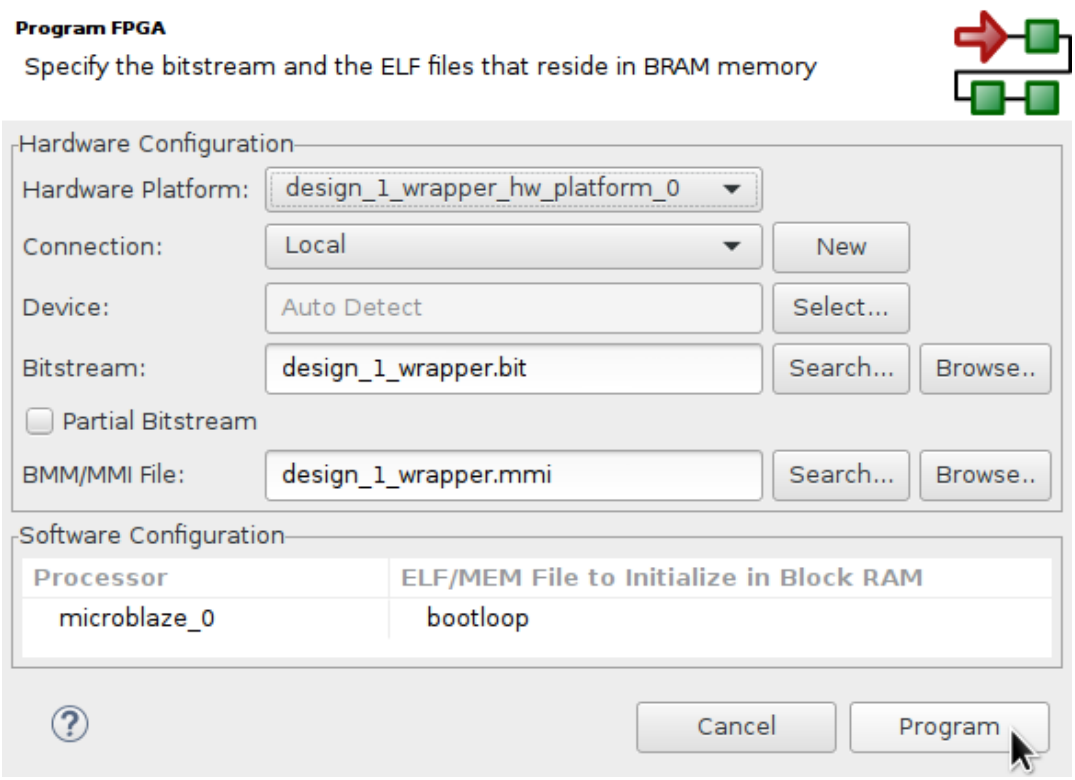
Les autres fichiers sont ceux définissant l'*address map*, identiques à ceux utilisés côté TLM.

9.6 Programmation du FPGA

Lancez la programmation du FPGA avec le bouton de la barre d'outils :



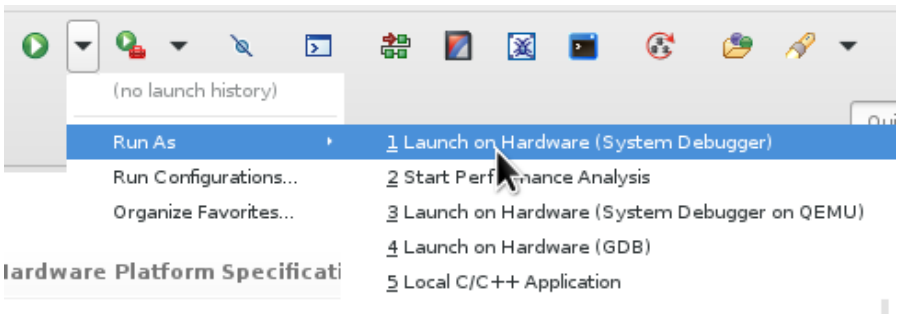
Gardez les paramètres par défaut et validez :



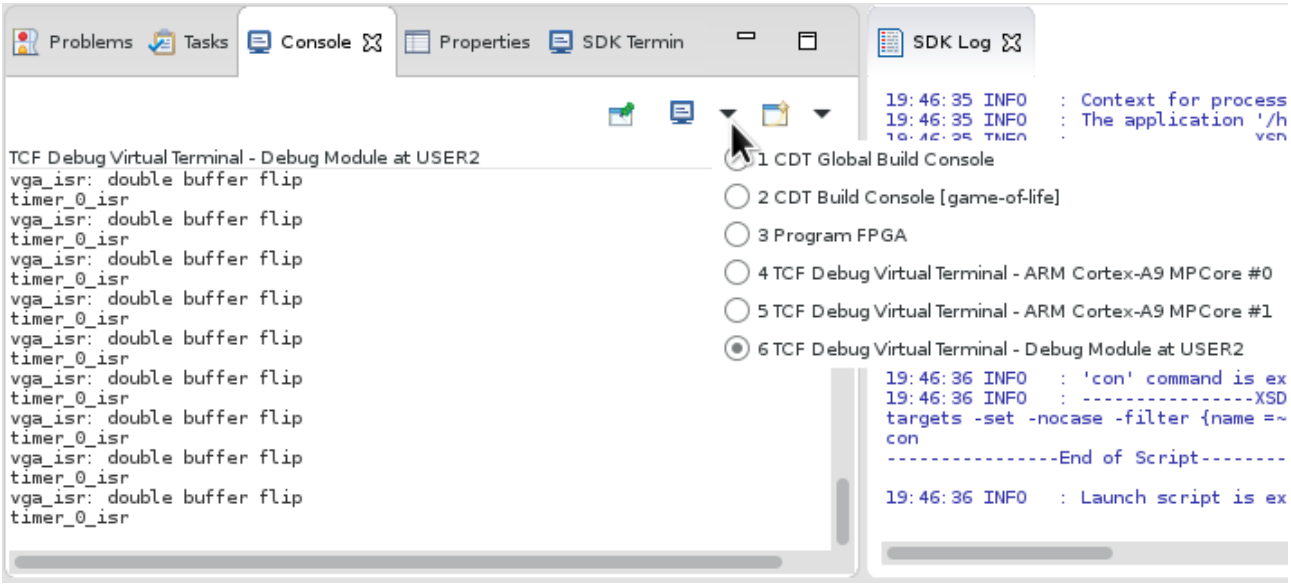
Pendant la programmation, vous aurez sans doute un avertissement « There is no PS in the design » (PS = Processing System), sans gravité : ignorez-le et validez avec « OK ».

Une fois le FPGA programmé, une mire (bandes noires et blanches verticales) apparaît à l'écran : c'est le motif par défaut de notre module VGA, qui n'est pas encore programmé, c'est normal !

Lancez maintenant l'application logicielle : dans « Project Explorer », sélectionnez « game-of-life », puis cliquez sur le bouton « Run As ... » et choisissez « Launch on hardware (System Debugger) » :



L'application s'exécute, le jeu de la vie doit apparaître à l'écran. On peut récupérer la sortie de printf directement depuis le SDK dans la fenêtre « console » (si besoin, choisissez la console « TCF Virtual Terminal, Debug Module ») :



Sur la carte FPGA, trouvez le bouton poussoir « btn3 (y16) » et appuyez dessus : une LED s'allume, et un glider supplémentaire doit apparaître à l'écran. Retrouvez dans le logiciel embarqué (`main.c`) la boucle d'ajout des gliders.

Vous pouvez essayer de modifier le logiciel embarqué, par exemple changer un message affiché par `printf`. Sauvegarder le fichier recompile l'application. Il n'est pas nécessaire de re-programmer le FPGA : cliquer sur « Run as » relance l'application.

9.7 Pour les curieux : les bugs du hardware

La partie qui suit n'est pas indispensable, mais si vous êtes curieux et que vous avez le goût du challenge, continuez à lire !

La plateforme matérielle est suffisante pour faire tourner notre logiciel, mais le contrôleur VGA comporte en réalité plusieurs bugs d'affichage.

Le premier est que la lecture en RAM n'est pas synchronisée explicitement avec l'affichage à l'écran. Par conséquents, il est possible que l'image soit décalée verticalement et/ou horizontalement. Ça n'arrive pas en pratique car tous les composants sont démarrées en même temps ... sauf si on ré-initialise la plateforme manuellement : appuyez sur le bouton BTN0(R18) pour le faire, et voyez l'image se décaler. Ce bug est bien identifié mais non-trivial à corriger.

Le second est que certains pixels sont affichés au mauvais endroit. On ne le voit pas sur le jeu de la vie où l'écran est essentiellement noir, mais essayez d'afficher une mire avec du noir et du blanc, et vous pourrez observer quelques portions de lignes mal affichées. Pour ce bug là, personne ne sait d'où il vient (sauf peut-être vous, chers lecteurs ?) !

10 Annexes

10.1 Origine des composants

Certains composants viennent du projet SocLib (`https://www.soclib.fr/`) :

— Le chargeur ELF (`elf-loader/`),

— L'ISS MicroBlaze (`microblaze.*`, `arithmetics.h`, `iss.h`, `register.h`),

— Le fichier `soclib_endian.h`.

La chaîne de compilation microblaze (gcc, ld, ...) est celle de Xilinx (`http://xilinx.wikidot.com/mb-gnu-tools`).

10.2 AXI Video Graphics Array (v1.00a) Data Sheet

Introduction

This document describes the specifications of the Video Graphics Array (VGA) core for the On-Chip AXI Bus.

The VGA is a 32-bit master-slave module that attaches to the AXI.

Features

— Supports 32-bit AXI v2.0 bus interface
— Monochrome (1 bit per pixel, 32-pixel per word)
— Supports 640 x 480 video resolution
— Generates 60 Hz vertical synchronization
— Configurable start address
— Support for interruptions and polling

Functional Description

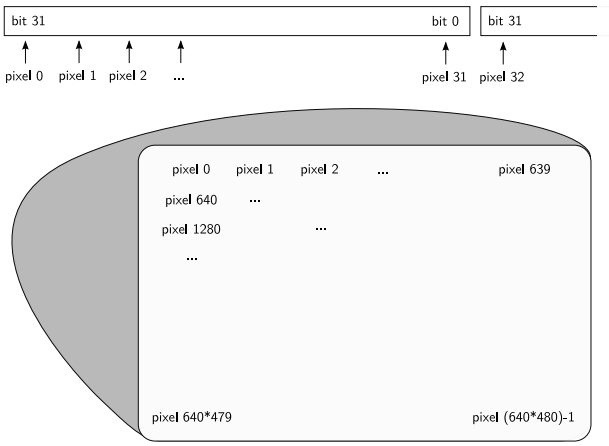


FIGURE 3 – Video buffer data format

After reset, the VGA is initially idle. User sets the start address by writing to the `CFG` register. If the value written to this register is different than `0x0`, then the VGA becomes enabled.

When enabled, the VGA continuously reads 32-bit words, starting from the start address (value of the `CFG` register), and incrementing until the end of the video buffer. It drives the color signals `RED`, `GREEN` and `BLUE` and the horizontal synchronization `HSYNC`.

Each time the VGA reaches the end, it drives the vertical synchronization signal `VSYNC` and sends an interruption using the `IP2INTC_Irpt` signal. It also sets the `INT` register to `0x1`. The `INT` register can be cleared by writing `0x1` to it.

The VGA can be idled by writing `0x0` to the `CFG` register.

Programming Model

Modes The VGA provides the following modes :
— IDLE : when `CFG` register is `0x0`. In this mode, a test pattern is displayed (vertical lines).
— ENABLED : otherwise

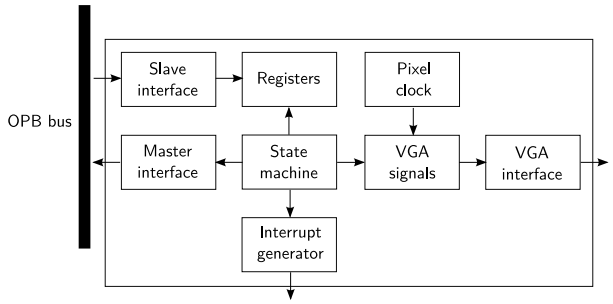


FIGURE 4 – VGA Block diagram

Register offsets

	Offset	Size	Type	Description
CFG	0x00	Word	R/W	Configuration reg.
STT	0x04	Word	R	Status register
INT	0x08	Word	R/W	Interrupt register

Registers descriptions

Configuration register The configuration registers holds the start address of the video buffer. The value should be modified soon after the vertical synchronization to give best results.

Status register The status register is not implemented.

Interrupt register The interrupt register is set to `0x1` after a vertical synchronization and can be cleared by writing `0x1` to it.