ENSEIRB-MATMECA

MISE EN ŒUVRE DU SOPC SUR COMPOSANTS FPGA INTEL ET AMD

Patrice NOUEL †
Patrice KADIONIK
kadionik.enseirb-matmeca.fr

TABLE DES MATIERES

1. BUT DES TRAVAUX PRATIQUES

Ces Travaux Pratiques ont pour but de présenter une approche au problème de la conception des SoC (*System On Chip*) dans ce qu'il a de particulier : mener conjointement le développement matériel et logiciel d'un projet.

Pour des raisons évidentes de souplesse d'utilisation, la plateforme matérielle est basée sur un circuit programmable FPGA transformant notre SoC en SoPC (*System On Programmable Chip*). Ceci permet d'obtenir des prototypes fonctionnels dans le minimum de temps dont on dispose.

Mais que doit-on trouver dans un enseignement sur les SoC ? Le système est composé d'éléments standards non originaux et bien connus : un processeur, des mémoires, des périphériques, une interface Ethernet… Tous ces éléments ont déjà fait l'objet d'enseignements spécifiques : cours microprocesseur, cours sur les réseaux, cours VHDL.
Dans tous ces enseignements, le matériel était parfaitement connu lorsqu'il s'agissait d'y associer du logiciel.

On le voit, ce qui va caractériser le SoC est en premier lieu sa capacité d'optimiser une solution en choisissant ce qui doit revenir au matériel et ce qui restera au logiciel et l'art de passer de l'un à l'autre. Créer ses propres périphériques ou ses propres instructions en VHDL, voilà en particulier ce que permet le SoC. On peut partir d'un processeur intrinsèquement peu performant et obtenir un système aux performances remarquables !

Les premiers TP de *codesign* ont été créés initialement en 2004 par Patrice Nouel (†1944-2022),

enseignant chercheur à l'ENSEIRB-MATMECA avec des cartes cibles Altera Stratix 1S10. Ils ont été ensuite mis à jour par moi avec des cartes cibles Intel DE10-Standard. De nouveaux TP ont été créés avec des cartes cibles Xilinx dans le but d'avoir un « grand » TP alliant l'intégration d'un périphérique matériel Libre dans un SoPC complété par la mise en œuvre de Linux embarqué pour le processeur *hardcore* Cortex-A9 du circuit FPGA Zynq. Un pilote de périphérique sous Linux sera alors développé pour pouvoir écrire l'application de test du périphérique sous Linux embarqué. De même, Xenomai sera mis en œuvre sur la cible pour mesurer des temps de latence sur système non chargé et système chargé.

On le voit, tout cela permet de réaliser LA synthèse de différents modules proposés dans l'option Systèmes Embarqués SE.

La mise en œuvre d'un système SoPC sur une carte Digilent à base de circuit FPGA Zynq et sur une carte Terasic à base de circuit FPGA Cyclone V a fait l'objet d'un sujet de « projets avancés » de l'option Systèmes Embarqués SE. Je tiens ainsi à remercier Maxime Gernet, Jean-Christophe Meyer, Ayoub Benyahya et Samir Mammeri de la promotion SE 2016-2017, Souleymane Soumah, Anis Yagoub et Fatima Ennaciri de la promotion SE 2022-2023 ainsi que Lucas Pallaro, Louis Lefebvre, Emile Vigneron et Yosr Ben Yahia de la promotion SE 2023-2024 pour leur travail et leur contribution à l'amélioration constante de l'enseignement de l'option SE…

Il est à noter que :

- Intel a racheté Altera en 2015.
- AMD a racheté Xilinx en 2022.

Par la suite, Altera sera synonyme d'Intel et Xilinx synonyme d'AMD…

Mots clés : SoPC, HLS, Intel, Altera, Xilinx, AMD, Quartus Prime, Vivado, Vitis HLS, *hardcore*, *softcore*, Cyclone V, NIOS II, Zynq, ARM, Cortex-A9, Linux, Linux embarqué, Xenomai, µC/OS II, langage C, VHDL

2. GRAND TP 1 : MISE EN ŒUVRE DU SOPC AVEC INTEL CYCLONE V

2.1. Introduction

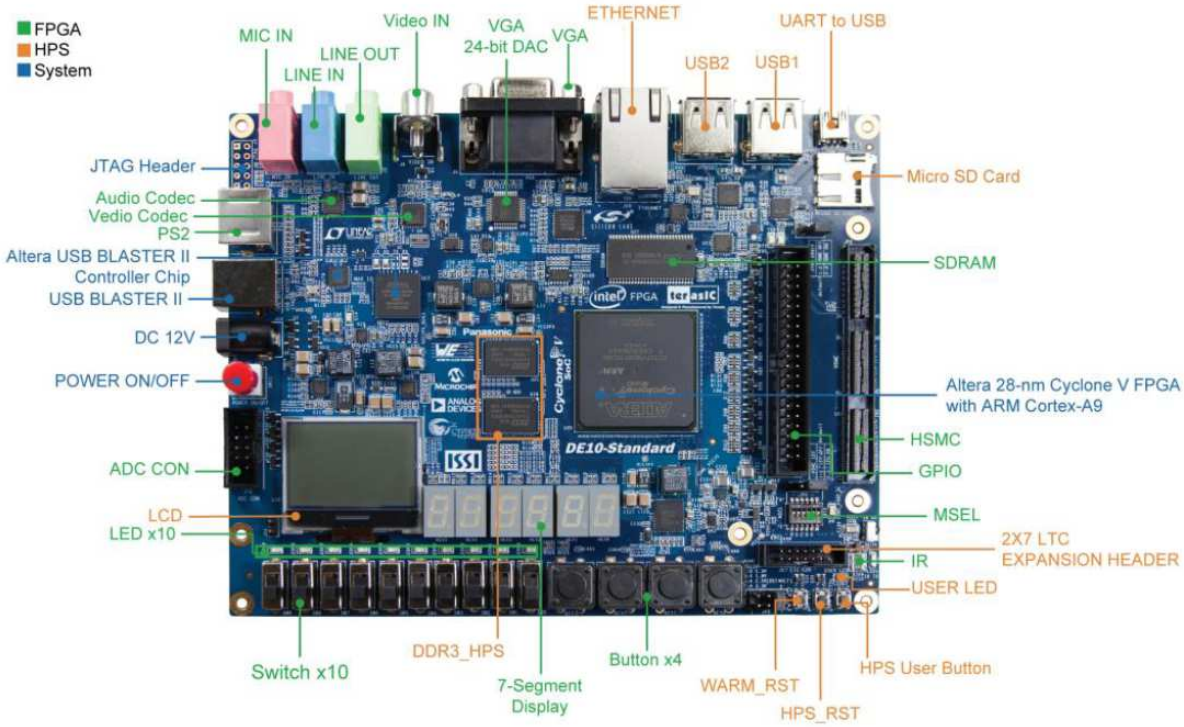
Il s'agit de mettre en œuvre l'environnement de développement SoPC d'Intel (ex Altera) qu'il s'agisse de *Platform Designer* pour la création d'un système SoPC (*System on Programmable Chip*), de *Quartus Prime* (édition standard) comme IDE (*Integrated Development System*) de placement routage et de synthèse ou bien de l'environnement *Eclipse* pour développement de la partie logicielle en langage C embarqué (ou mode dit *bare metal*).

Ce premier « grand » TP se décompose en :

- La création pas à pas d'un système SoPC à base du processeur *softcore* NIOS II pour circuit FPGA Cyclone V® avec les outils Intel *Quartus Prime* et *Platform Designer*.
- La mise en œuvre du langage C avec *Eclipe* sur le processeur NIOS II du circuit FPGA Cyclone V.
- Le développement du BSP (*Board Support Package*) avec le langage C embarqué pour piloter de façon simple les périphériques du système SoPC.
- Le développement d'applications avec le langage C en mode *bare metal* et mise en œuvre du noyau Temps Réel µC/OS II.

2.2. Carte cible Terasic DE10-Standard

La carte cible mise en œuvre pour le SoPC sur circuit FPGA Cyclone V d'Intel est une carte d'évaluation Terasic DE10-Standard.



Carte cible Terasic DE10-Standard

La carte DE10-Standard intègre un circuit FPGA Cyclone V qui incorpore un processeur *hardcore* ARM dans la partie HPS (*Hard Processor System*) et une zone de programmation logique PL (*Programmable Logic*). Cette approche duale existe aussi chez AMD avec son circuit FPGA Zynq.

La carte possède les caractéristiques suivantes pour la partie PL :

- Circuit FPGA Intel Cyclone V SE 5CSXFC6D6F31C6N.
- Mémoire pour la configuration de la partie PL EPCS128.
- Sonde USB-Blaster II pour la programmation avec le mode JTAG.
- 64 Mo de SDRAM.
- 4 boutons poussoir.
- 10 switchs.
- 10 leds rouges.
- 6 afficheurs 7 segments.
- Codec audio 24 bits avec line-in, line-out et entrée microphone.
- Convertisseur CNA 8bits (x3) avec sortie VGA.
- Décodeur TV (NTSC/PAL/SECAM) et entrée TV.
- Connecteur PS/2.
- Emetteur/récepteur IR.
- Convertisseur CAN avec interface SPI.

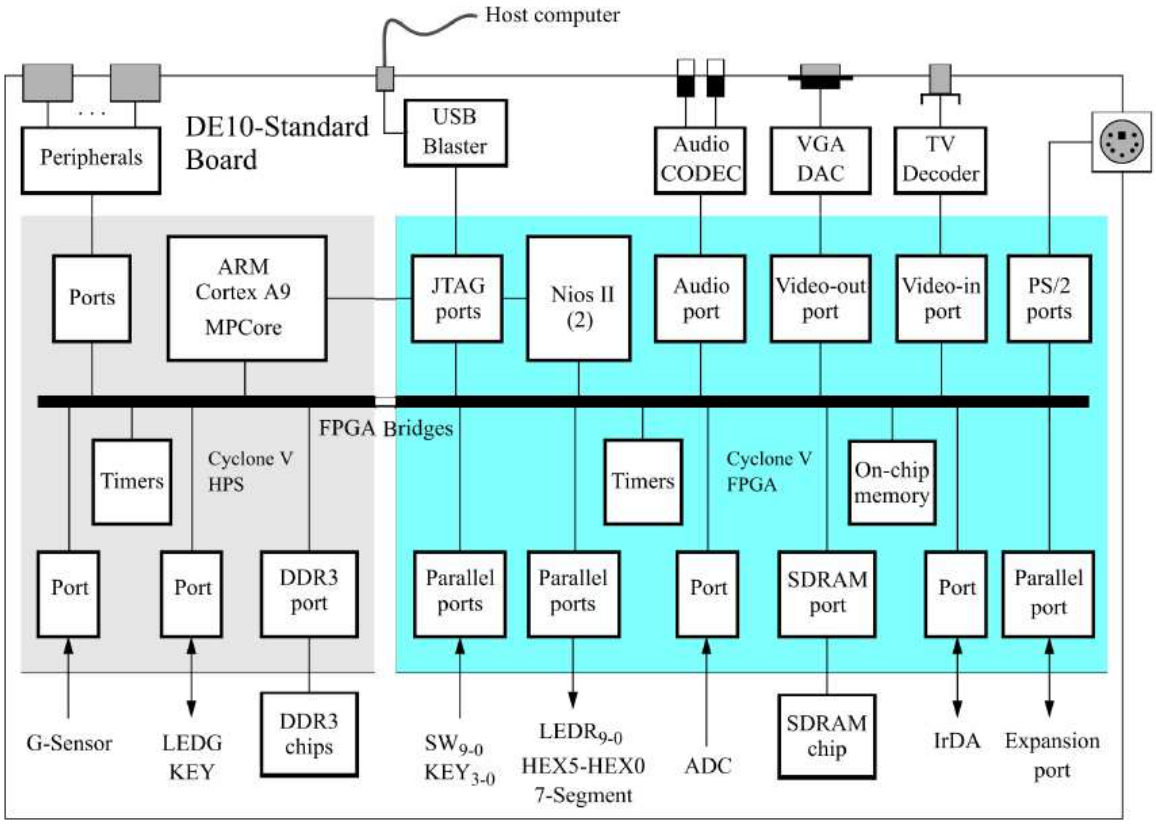
La carte possède les caractéristiques suivantes pour la partie HPS :

- Processeur double cœur ARM Cortex-A9 à 925 MHz.
- 1 Go de SDRAM DDR3.
- 1 interface Ethernet Gb/s.
- 2 ports USB Host.
- 1 *socket* micro SD.
- 1 accéléromètre avec interface I2C.
- 1 connecteur UART ver USB Mini-B.
- Boutons *warm reset* et *cold reset*.
- 1 bouton utilisateur et 1 led utilisateur.
- 1 module LCD 128x64 points.

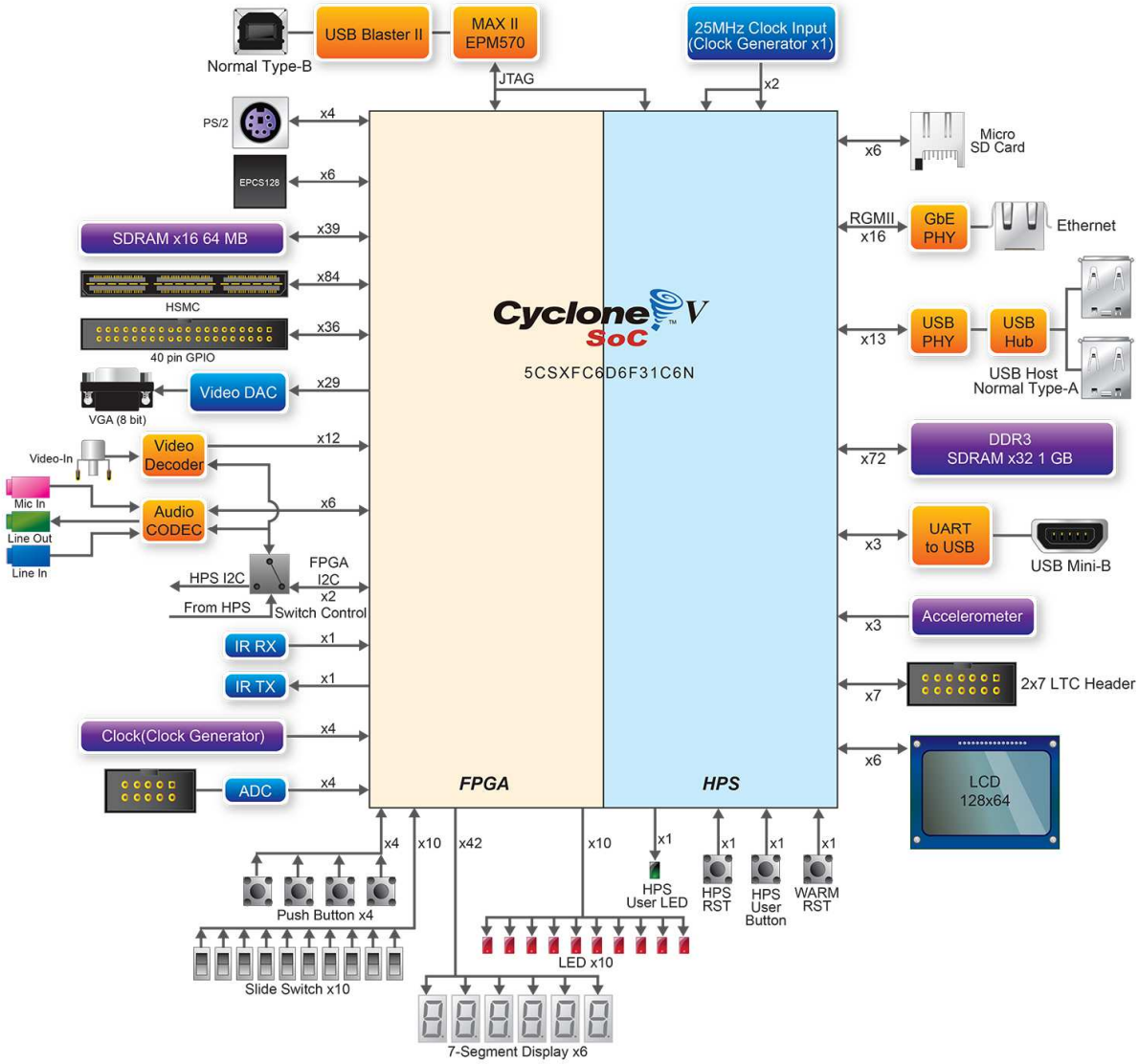
Il faut noter que l'**on ne peut pas atteindre directement des périphériques de la partie HPS depuis la partie PL et inversement sans la mise en œuvre de ponts** (*bridge*). Par exemple, on ne peut pas piloter directement depuis la partie PL le module LCD 128x64 points.

La figure suivante présente l'ensemble des périphériques accessibles sur la carte cible DE10-Standard. Cela correspond à un *design* de référence d'un système SoPC appelé *Computer System* par Intel mettant en œuvre 2 processeurs *softcore* NIOS II et le processeur ARM.

Nous n'allons pas partir de ce *design* de référence trop complexe et tout fait mais nous allons développer notre propre *design* SoPC à base d'un seul processeur NIOS II pour apprendre à maîtriser les outils Intel.



***Design* de référence *Computer System* pour la carte cible Terasic DE10-Standard**



Périphériques de la carte cible Terasic DE10-Standard

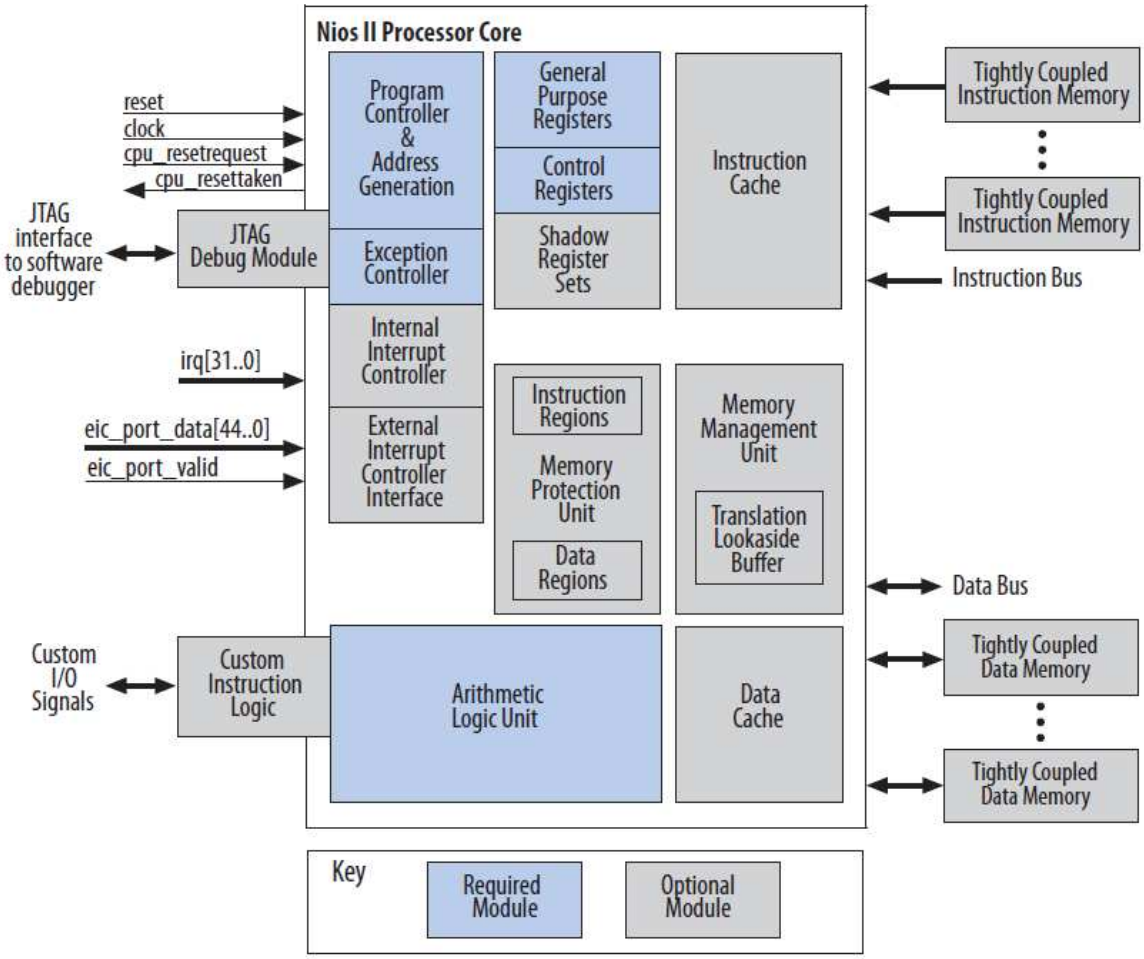
2.3. Présentation du processeur softcore NIOS II d'Intel

Le processeur NIOS II (2ème génération du processeur NIOS) est un processeur RISC *softcore* entièrement synchrone, son architecture interne étant de type Harvard. Il possède au maximum 6 niveaux de *pipeline*, cadencé à quelques dizaines de MHz, avec une largeur de bus de 32 bits. Ses performances vont jusqu'à 250 MIPS (*Million Instructions per Second*).

Les caractéristiques du processeur NIOS II sont :

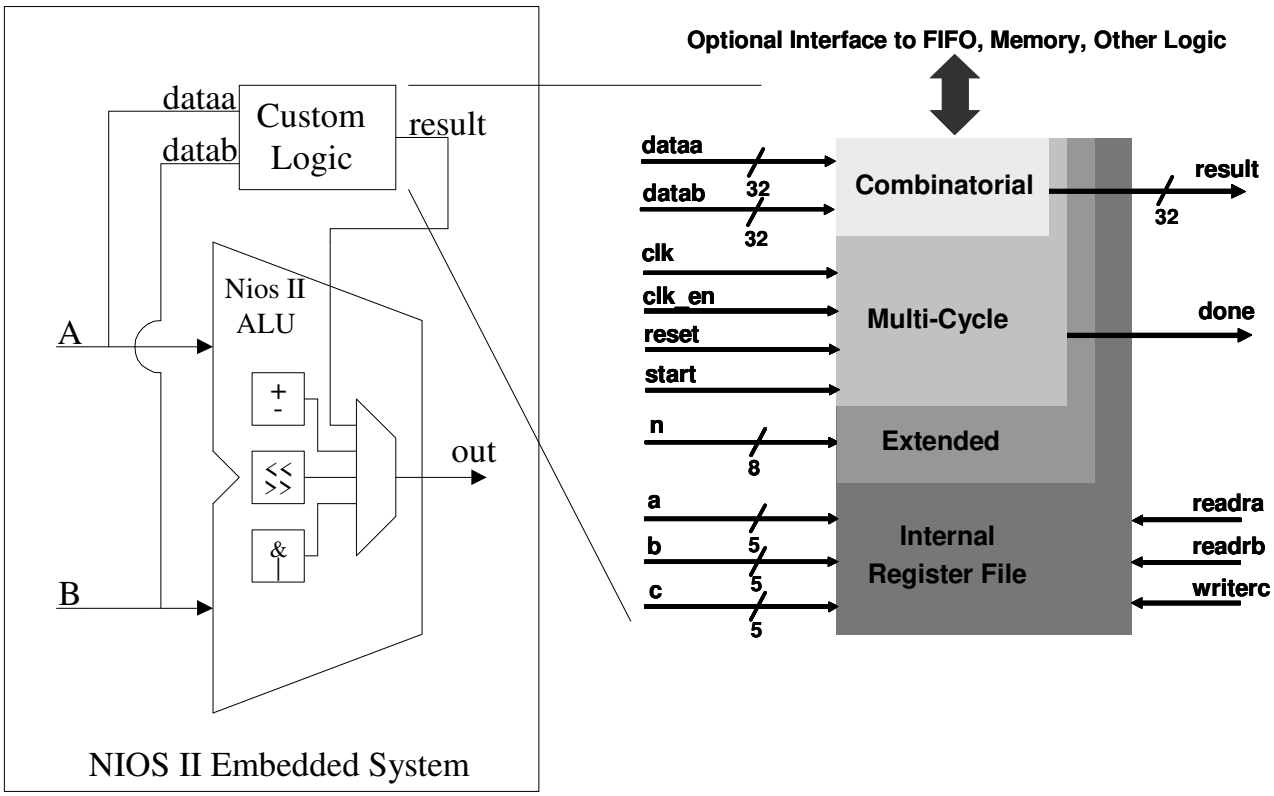
- Architecture RISC.
- Jeu d'instructions 32 bits.
- 32 registres généraux.
- 32 sources d'interruption.
- Instruction assembleur pour multiplications et divisions entières 32x32 bits pour un résultat 32 bits.
- Instructions pour multiplications 64 et 128 bits
- Instructions optionnelles pour opérations sur nombres réels simple précision.

- Accès à une variété de périphériques *on-chip* et interfaces vers les périphériques et la mémoire *off-chip*.
- Module de *debug* matériel.
- MMU (*Memory Management Unit*) optionnelle.
- MPU (*Memory Protection Unit*) optionnelle.



Architecture du processeur NIOS II

Il est possible d'accélérer certains traitements en ajoutant des instructions personnelles ou C*ustom Instructions* (décrites en langage VHDL ou Verilog) au processeur NIOS II. De cette manière, il est possible de réaliser la surcharge d'opérateurs ou simplement d'étendre le jeu d'instructions.



Instruction personnalisée avec le processeur NIOS II

Lors de la configuration du processeur NIOS II avec l'outil *Platform Design* (ex outil *SoPC Builder* ou *Qsys*), il est possible de choisir entre 2 versions du processeur NIOS II : une première version *Economy* qui utilise moins de surface de silicium du composant FPGA et une version *Fast* qui est la plus rapide mais plus consommatrice de ressources.

	NIOS II /f	**NIOS II /e**
Pipeline	6 niveaux	Non
Multiplication Matériel	1 cycle	Par logiciel
Branch Prediction	Dynamic	Non
Cache d'Instructions	Configurable	Non
Cache de données	Configurable	Non
Instructions Personnalisés	< 256	

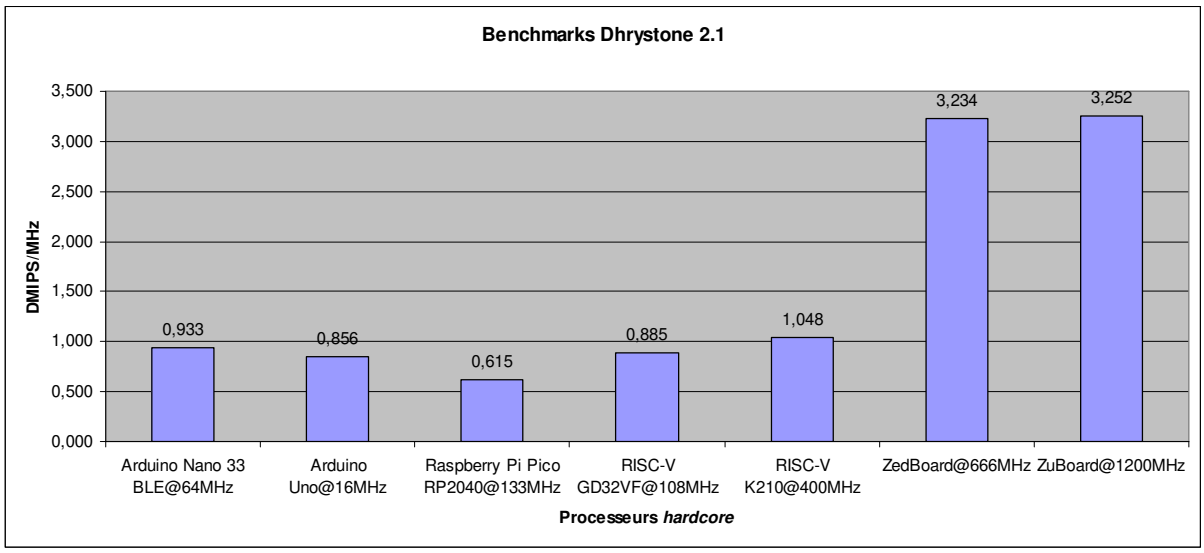
Les 2 versions du processeur NIOS II

Le tableau suivant présente les performances en DMIPS *(Dhrystone Million Instructions per Second*) à l'aide du *benchmark* Dhrystone du processeur NIOS II sur les différentes familles de composants FPGA d'Intel (Agilex, Stratix V, Arria 10, Cyclone V, MAX 10…).

Métrique	NIOS II/f	NIOS II/e
DMIPS/MHz	0.753	0.107

Performances du processeur *softcore* NIOS II

Pour comparaison, le tableau suivant donne les performances de quelques plateformes matérielles à base de processeurs *hardcore* (mesurées par l'auteur) à l'aide du *benchmark* Dhrystone 2.1 :



Exemples de performances de processeurs *hardcore*

Notons que dans la phase de construction du circuit avec l'outil *Platform Designer*, il est possible d'inclure différents périphériques standards en utilisant le bus multimaître *Avalon* du processeur NIOS II :

- Mémoire.
- *Timer*.
- Liaison série JTAG/UART.
- Interface écran VGA.
- E/S parallèles.
- Interface Ethernet.
- JTAG.
- …

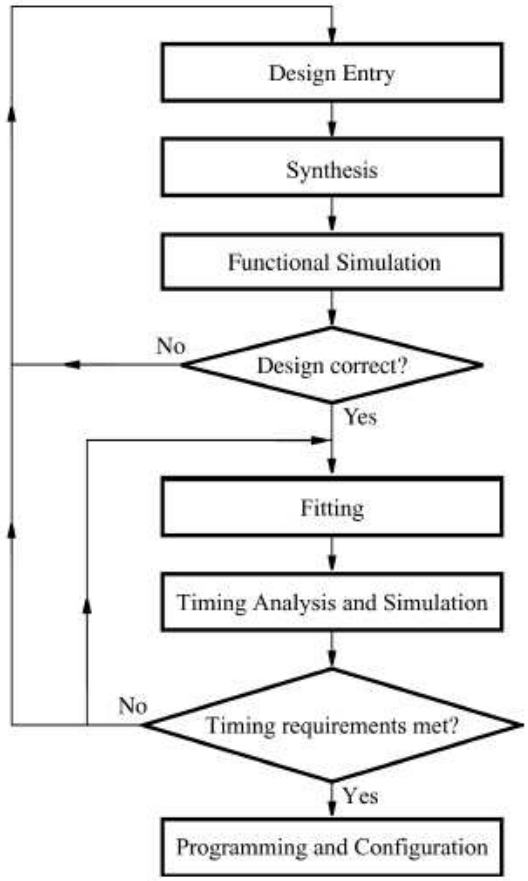
3. EX 1 INTEL : CONSTRUCTION DU DESIGN DE REFERENCE

3.1. Introduction

Il s'agit d'utiliser l'environnement de développement SoPC d'Intel qu'il s'agisse de *Quartus Prime* et de *Platform Designer* pour la construction du *design* de référence donc du système SoPC.

Après synthèse, le circuit SoPC sera programmé dans le circuit FPGA de la carte DE10-Standard.

La figure suivante récapitule la démarche de conception typique d'un système SoPC :



Conception d'un système SoPC

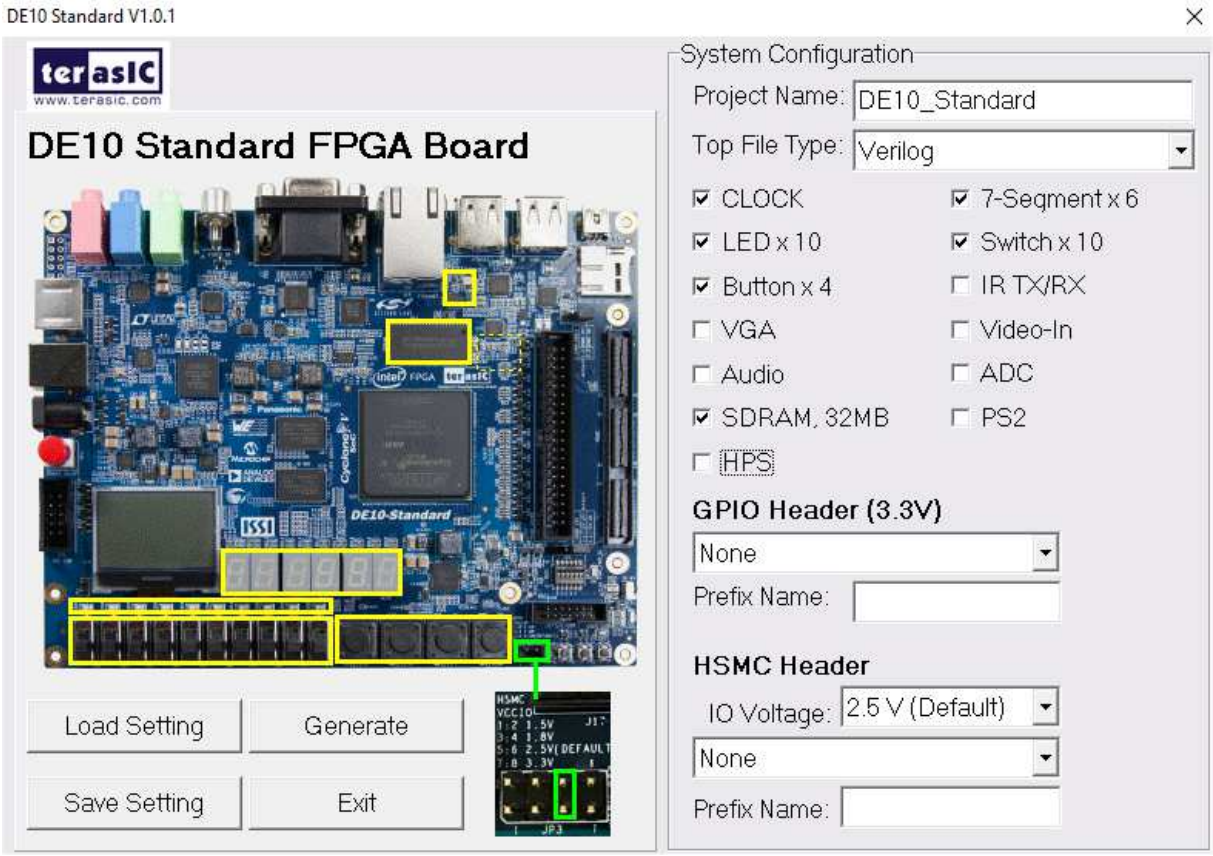
Les différentes phases sont (analogues à la conception d'un système numérique) :

- Description du système SoPC : on utilisera l'outil graphique *Platform Designer*. Le code généré sous-jacent est soit du VHDL ou soit du Verilog.
- Synthèse : le système SoPC est analysé puis synthétisé pour viser un circuit FPGA, ici le circuit FPGA Cyclone V.

- Simulation fonctionnelle : vérification des délais de propagation et métastabilité.
- Placement/routage (*fitting*) : analyse temporelle : les délais de propagation dans les circuits sont analysés afin d'amener des indications sur la performance du circuit.
- Simulation temporelle : après placement/routage, le circuit est simulé en prenant en compte cette fois-ci les contraintes temporelles.
- Programmation : le circuit FPGA est programmé avec la sonde USB-Blaster à partir du fichier de programmation `.sof` (`.bit` chez AMD).

Intel fournit un outil Windows *DE10_Standard_SystemBuilder* qui permet de générer les fichiers nécessaires pour démarrer le projet *Quartus Prime*.

Le *design* est référence qui est un sous-ensemble du *design Computer System* est présenté sur la figure suivante.



***Design* de référence *DE10_Standard_golden_top* du système SoPC**

Outre le processeur NIOS II (version *fast*), le système SoPC contient les périphériques externes au circuit FPGA suivants de la carte cible DE10-Standard :

- Mémoire SDRAM de 64 Mo.
- 10 leds.
- 4 boutons poussoir.
- 6 afficheurs 7 segments.
- 10 switchs.

On rajoutera aussi les périphériques internes au circuit FPGA suivants :
- Liaison série JTAG/UART.
- *Timer* 32 bits.

On aura ainsi au démarrage 5 fichiers pour notre projet *Quartus Prime* :
- Fichier `.qpf` (*Quartus Project File*) : fichier projet à ouvrir avec *Quartus Prime*.
- Fichier `.qsf` (*Quartus Setting File*) : fichier de configuration avec les paramètres et l'affectation des broches du projet.
- Fichier `.v` : fichier *Top-Level* du plus haut niveau hiérarchique en langage Verilog.
- Fichier `.sdc` (*Synopsis Design Constraints*) : fichier des contraintes temporelles.
- Fichier `.htm` : fichier HTML donnant sous forme de tableaux les assignements des broches du circuit FPGA et le nom des signaux du *design* SoPC.

On notera que le nom des signaux Verilog (notamment dans le fichier *Top-Level* `.v`) est le lien dans tout le projet SoPC notamment sur l'usage des broches du circuit FPGA.

Le nom de notre projet *Quartus Prime* est *DE10_Standard_golden_top*.

On aura donc les 5 fichiers suivants :
- Fichier `DE10_Standard_golden_top.qpf` : projet *Quartus Prime*.
- Fichier `DE10_Standard_golden_top.qsf`.
- Fichier `DE10_Standard_golden_top.v` : fichier Verilog *Top-Level*.
- Fichier `DE10_Standard_golden_top.sdc`.
- Fichier `DE10_Standard_golden_top.htm`.

Le contenu du fichier `Verilog DE10_Standard_golden_top.v` est le suivant :

```
module DE10_Standard_golden_top(

      ///////// CLOCK /////////
      input              CLOCK2_50,
      input              CLOCK3_50,
      input              CLOCK4_50,
      input              CLOCK_50,

      ///////// KEY /////////
      input    [ 3: 0]   KEY,

      ///////// SW /////////
      input    [ 9: 0]   SW,

      ///////// LED /////////
      output   [ 9: 0]   LEDR,

      ///////// Seg7 /////////
      output   [ 6: 0]   HEX0,
      output   [ 6: 0]   HEX1,
      output   [ 6: 0]   HEX2,
      output   [ 6: 0]   HEX3,
      output   [ 6: 0]   HEX4,
      output   [ 6: 0]   HEX5,

      ///////// SDRAM /////////
      output             DRAM_CLK,
      output             DRAM_CKE,
```

```
        output    [12: 0]   DRAM_ADDR,
        output    [ 1: 0]   DRAM_BA,
        inout     [15: 0]   DRAM_DQ,
        output              DRAM_LDQM,
        output              DRAM_UDQM,
        output              DRAM_CS_N,
        output              DRAM_WE_N,
        output              DRAM_CAS_N,
        output              DRAM_RAS_N
);

endmodule
```

Le nom du projet *Quartus Prime* est bien *DE10_Standard_golden_top*.

Les périphériques extérieurs au circuit FPGA sont connectés aux broches du circuit FPGA par leur nom :

- `KEY` : 4 boutons poussoir. Bus de 4 signaux.
- `SW` : 10 switchs. Bus de 10 signaux.
- `LEDR` : 10 leds rouges. Bus de 10 signaux.
- `HEXn` : 6 afficheurs 7 segments. 6 bus de 7 signaux.
- `CLOCKx_xx` : horloges.
- `DRAM_` : signaux de contrôle de la SDRAM externe.

Si l'on regarde le fichier `DE10_Standard_golden_top.qsf`, on retrouve l'association (nom_du_signal <-> numéro de broche).
Par exemple, le bouton poussoir `KEY0` est connecté à la broche du circuit FPGA AJ4 (`set_location_assignment PIN_AJ4 -to KEY[0]`).

3.2. Ajout du processeur NIOS II et de ses périphériques

Par la suite, on adoptera les conventions suivantes :
Commande Linux PC hôte :

```
host% commande Linux
```

Commande Linux PC hôte pour le développement Intel :

```
[NiosII EDS]$
```

- Démarrer le PC sous Linux. Se connecter sous le nom **se01**, mot de passe : **se01** ☺ pour le groupe 1 et sous le nom **se02**, mot de passe : **se02** ☺ pour le groupe 2.

- Se placer dans son répertoire de travail :
  `host% cd`

- Recopier le fichier `tp-de10.tgz` sous `/home/kadionik/` :
  `host% cp /home/kadionik/tp-de10.tgz .`

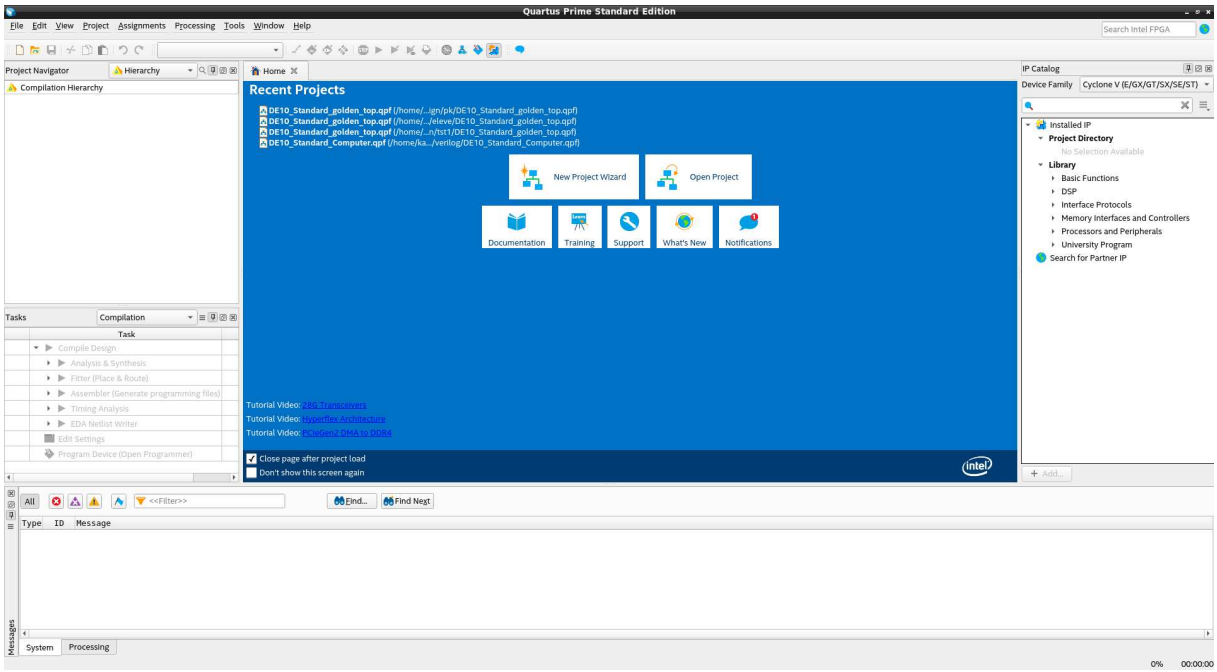
- Décompresser et installer le fichier `tp-de10.tgz` :
  `host% tar -xvzf tp-de10.tgz`

- Se placer ensuite dans le répertoire `de10/`. **L'ensemble du travail sera réalisé à partir de ce répertoire ! Les chemins seront donnés par la suite en relatif par rapport à ce répertoire...**
```
host% cd de10
```

- Se placer dans le répertoire `standard_nios/`. Nous allons maintenant construire notre *design* de référence *DE10_Standard_golden_top* :
```
host% cd standard_nios
host% ls
DE10_Standard_golden_top.htm  DE10_Standard_golden_top.sdc
DE10_Standard_golden_top.qpf  DE10_Standard_golden_top.v
DE10_Standard_golden_top.qsf  ip/
```

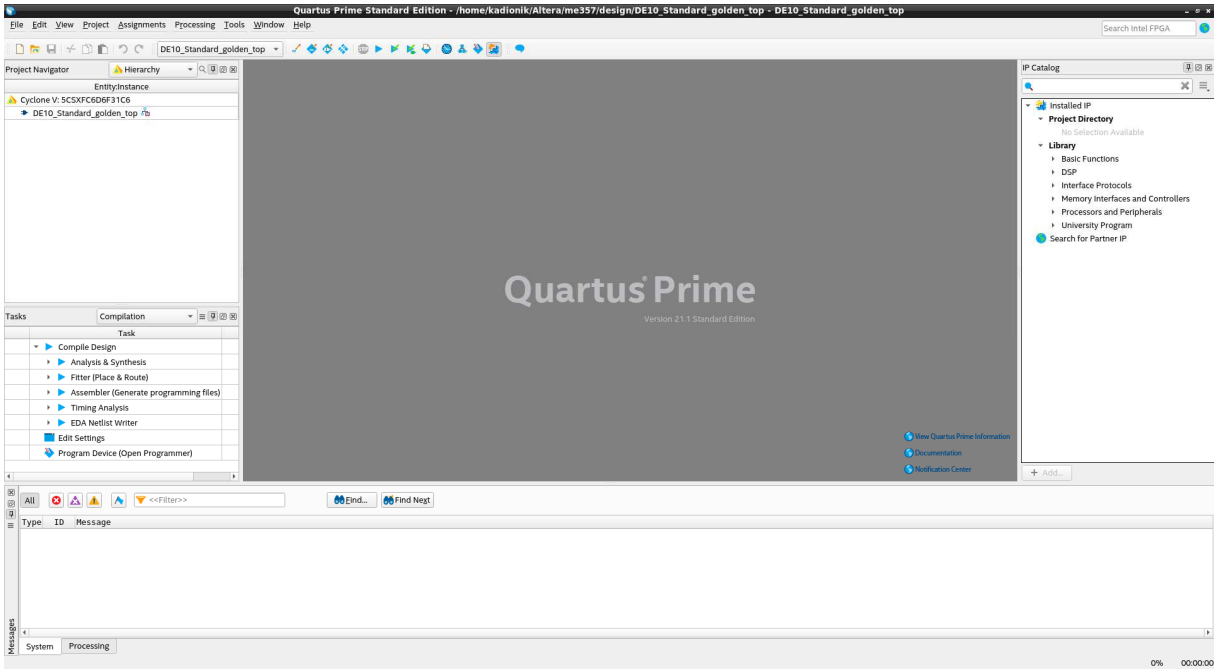
- Se placer dans l'environnement de développement Intel puis lancer *Quartus Prime* :
```
host% n2sdk
[Xilinx EDK]$ quartus
```

On obtient la figure suivante qui est l'interface graphique de *Quartus Prime* :



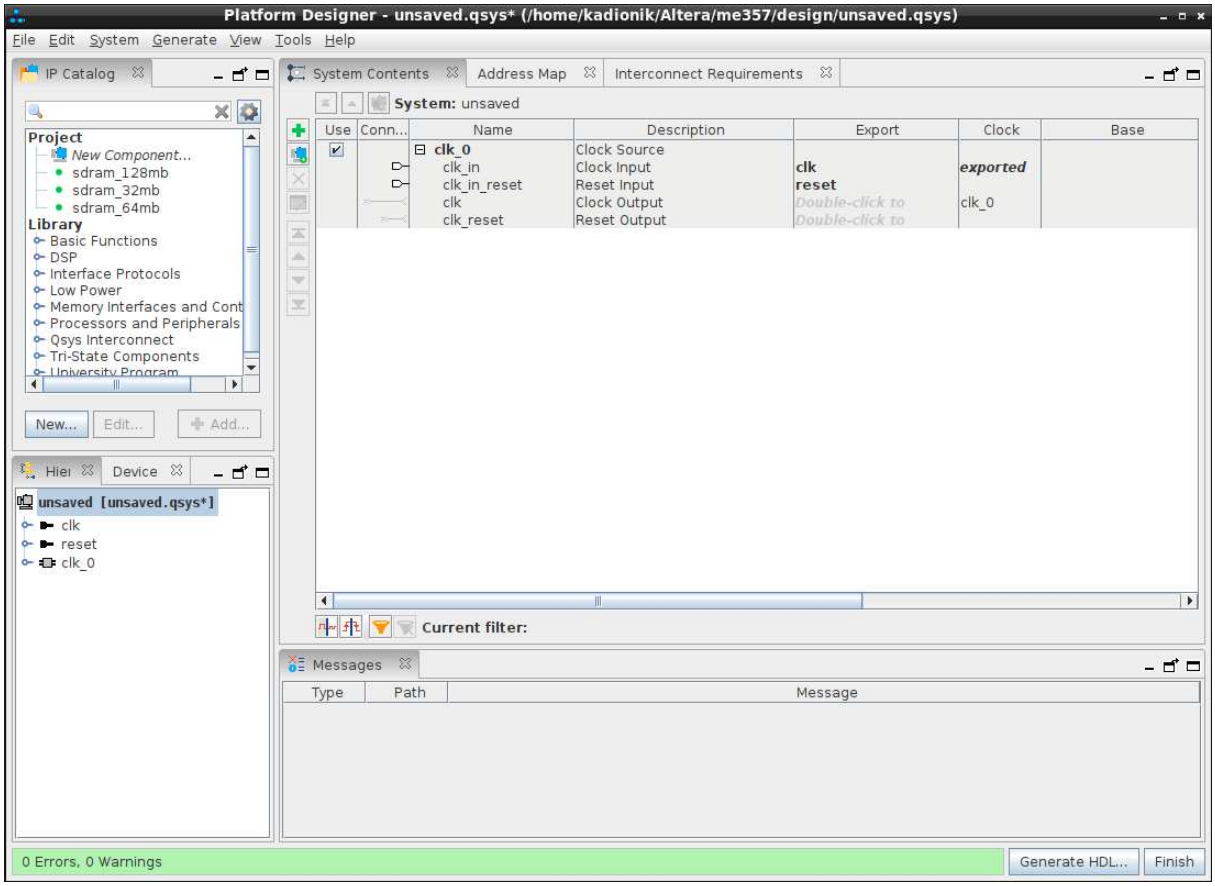
Interface graphique de *Quartus Prime* (1)

- Ouvrir le projet *Quartus Prime DE10_Standard_golden_top* par le menu *File > Open Project...* (ouverture du fichier `DE10_Standard_golden_top.qpf`). On obtient la figure suivante :



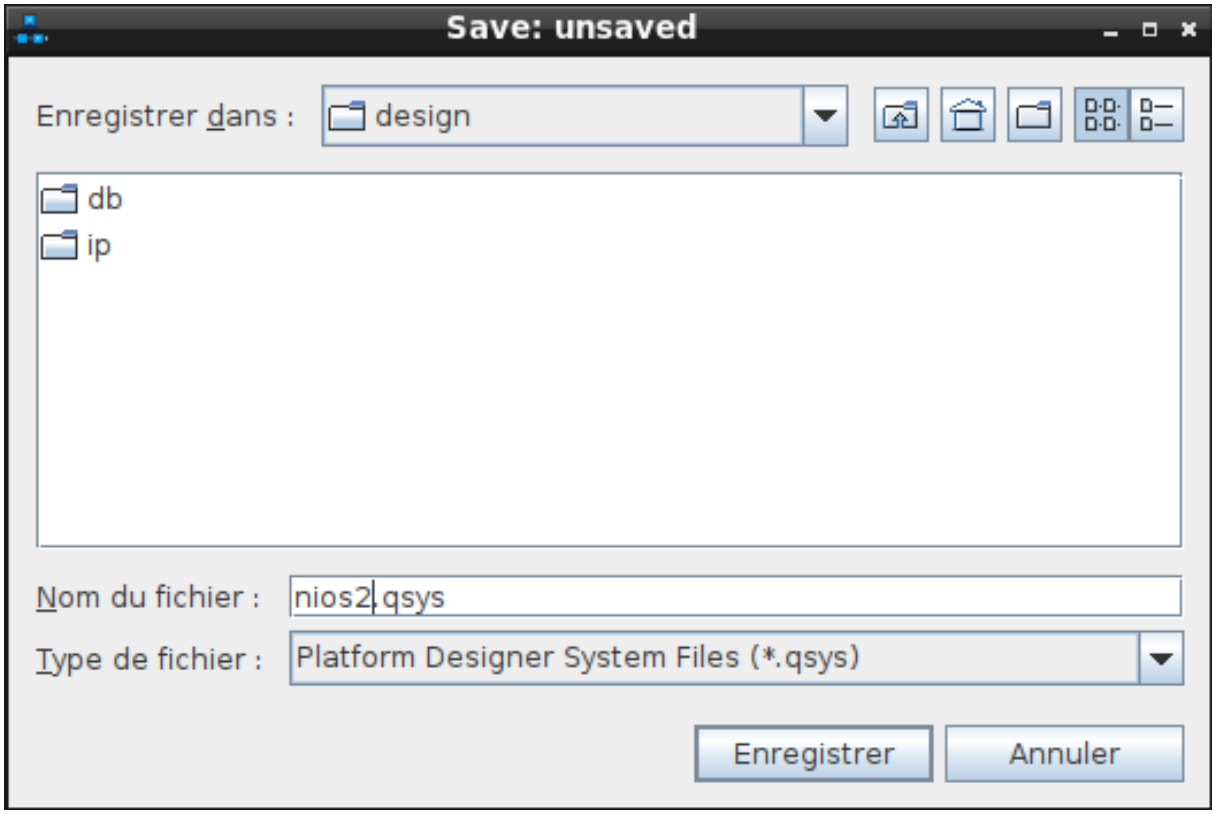
Projet *DE10_Standard_golden_top* (2)

- Lancer ensuite *Platform Designer* par le menu *Tool > Platform designer*. On obtient la figure suivante :



Interface graphique de *Platform Designer* (3)

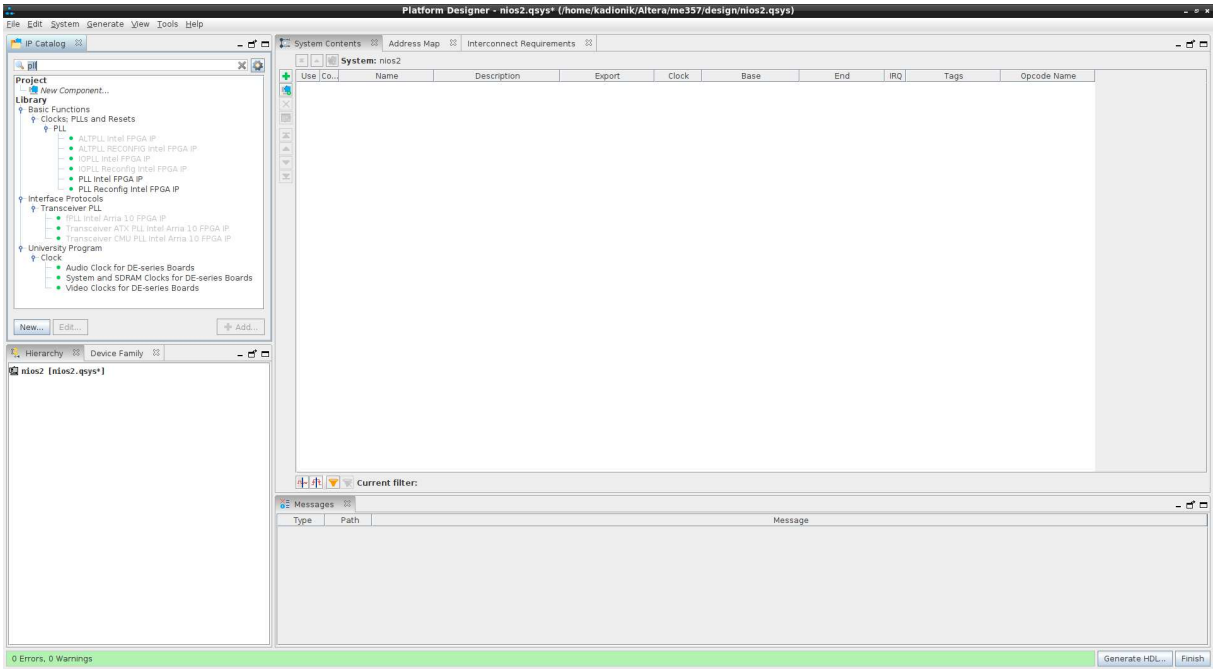
- Sauvegarder le projet *Platform Designer* en le nommant `nios2.qsys` par le menu *File > Save*. On obtient la figure suivante :



Sauvegarde du projet *Platform Designer* nios2.qsys (4)

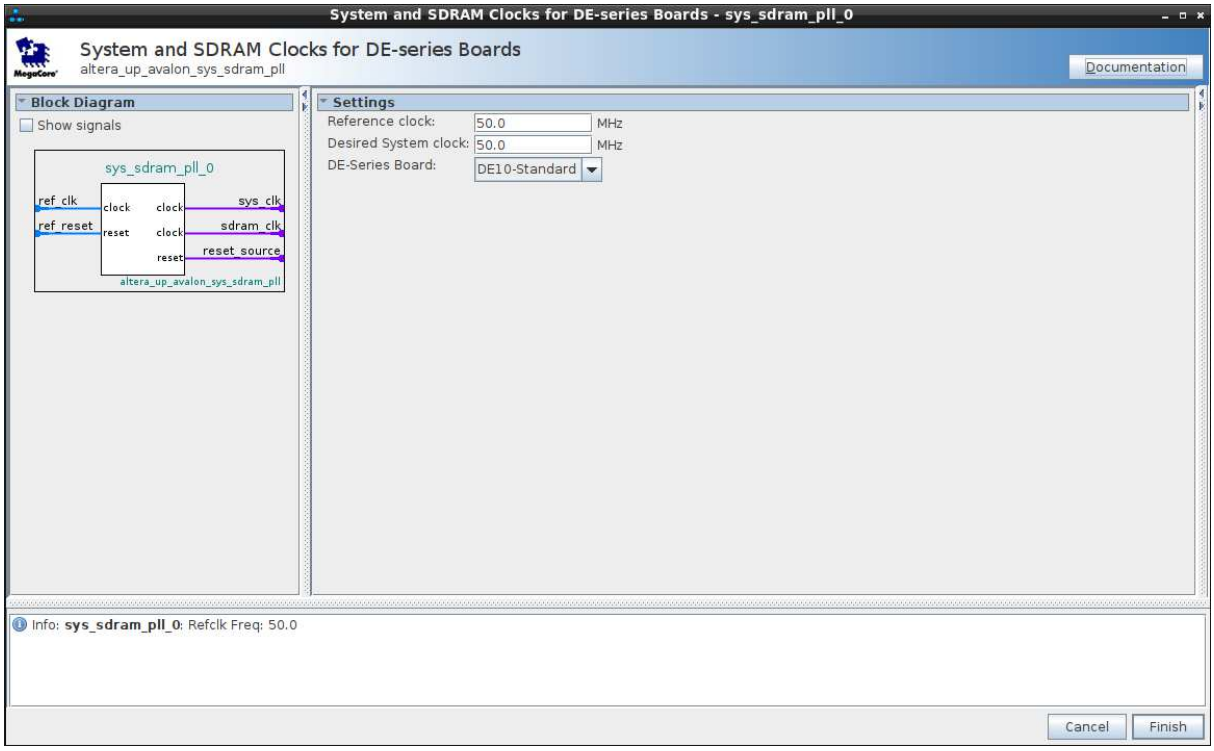
- Sélectionner et effacer le bloc IP (*Intellectual Property*) *clock_0*.

- Dans la fenêtre *IP Catalog*, rechercher les occurrences contenant la chaîne de caractères pll. On obtient la figure suivante :



Blocs IP contenant l'occurrence pll (5)

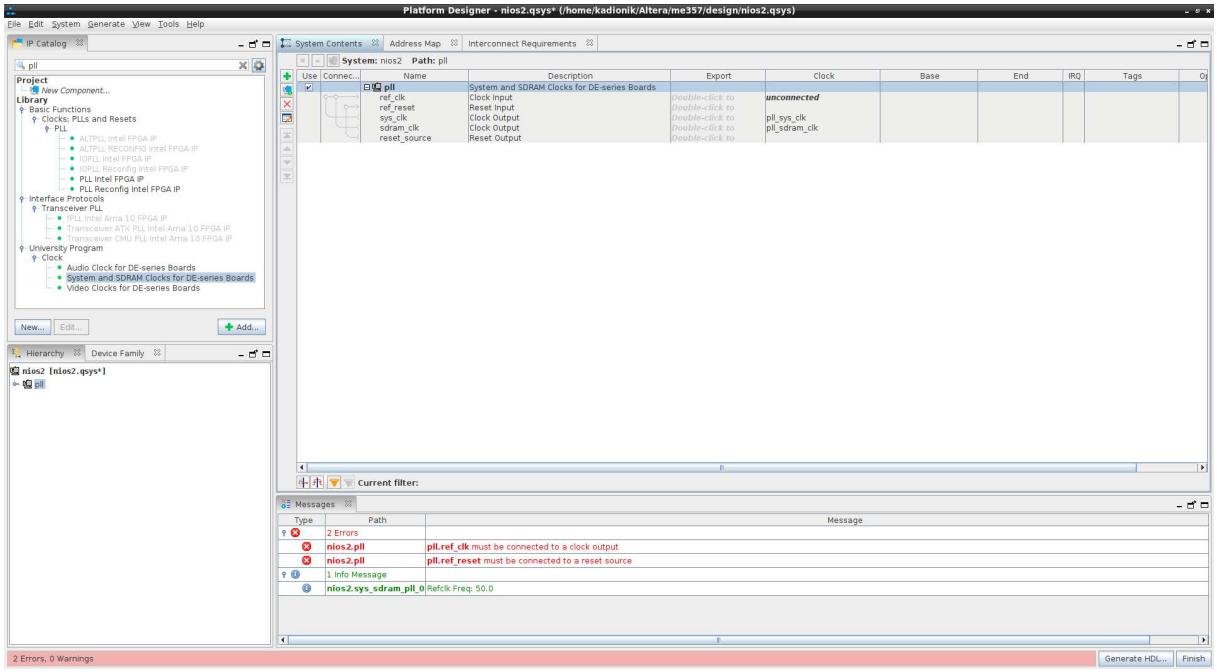
- Sélectionner le bloc IP *System and SDRAM Clocks for DE-series Boards*. On obtient la figure suivante :



Configuration du bloc IP *System and SDRAM Clocks for DE-series Boards* (6)

On ajustera le champ *Desired System clock* à 100.0 MHz. On ne touchera pas à la valeur par défaut des autres champs. Puis on cliquera sur le bouton *Finish*.

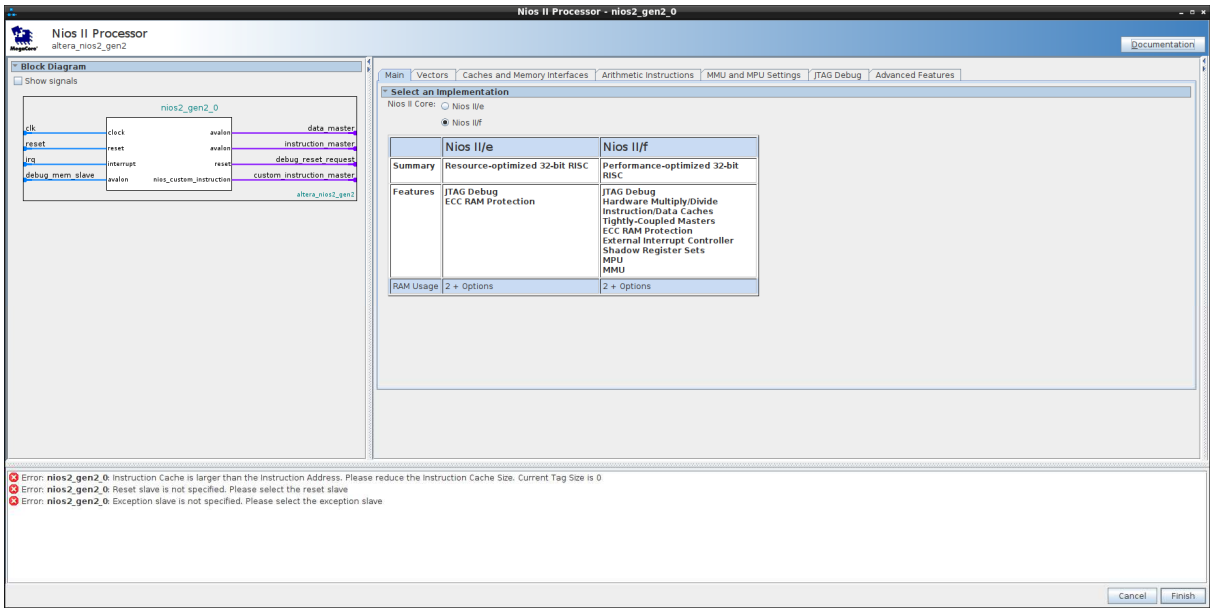
- Changer le nom du bloc IP *sys_sdram_pll_0* en *pll*. Pour cela, on se positionne sur le label *sys_sdram_pll_0*, puis avec le clic droit de la souris, on accède au menu contextuel et au choix *Rename*. On obtient la figure suivante :



Bloc IP *pll* (7)

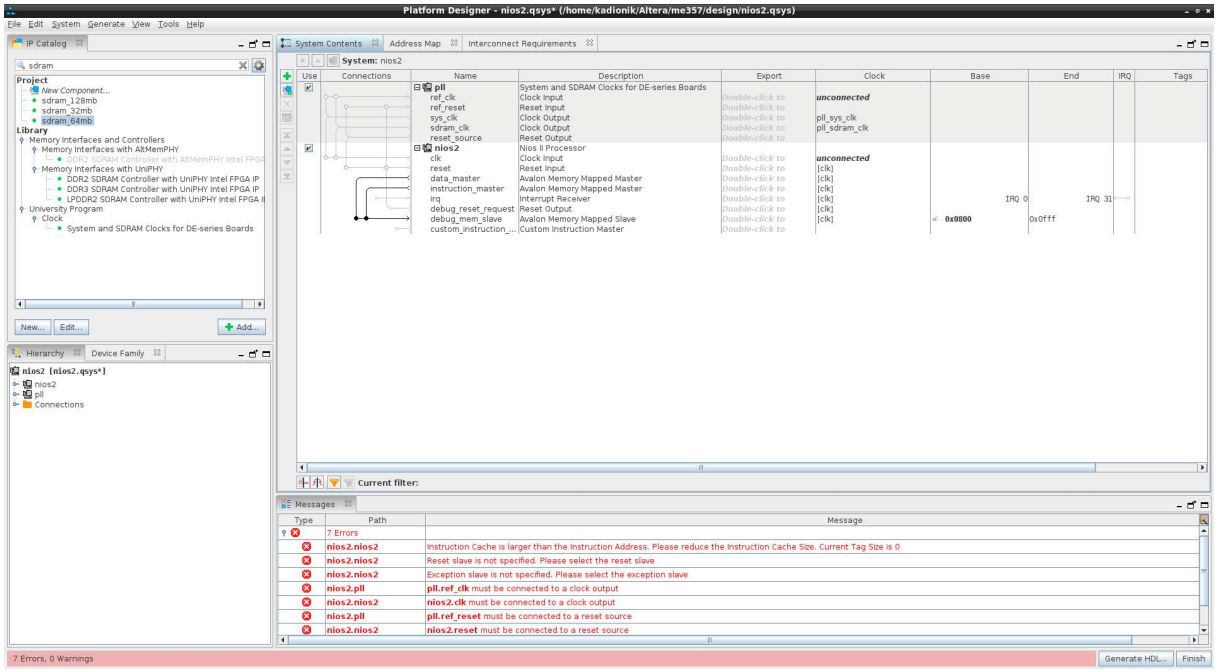
La sélection et la configuration du premier bloc IP ont été détaillées. Pour les suivants, on adoptera le même principe.

- Ajouter le bloc IP *NIOS II*. Dans la fenêtre *IP Catalog*, rechercher les occurrences commençant par nios. On choisira le bloc IP *Nios II Processor*. On choisira la version *fast*. On obtient la figure suivante :



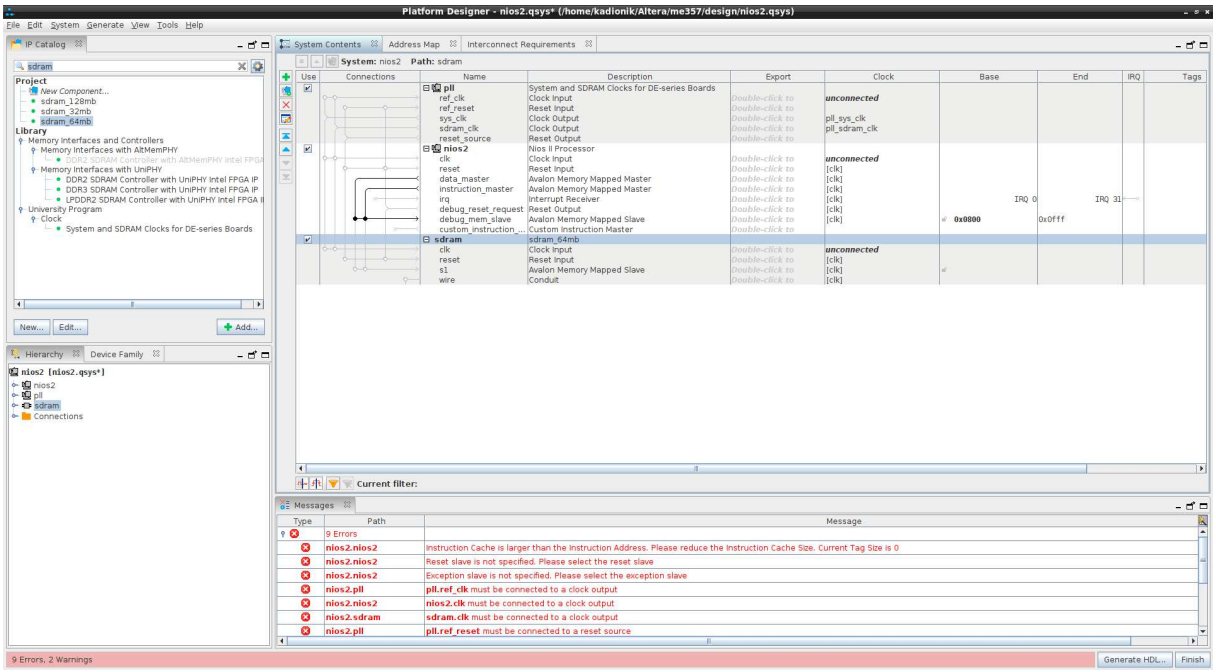
Bloc IP *Nios II Processor* (8)

- Renommer le bloc IP *nios2*. On obtient la figure suivante :



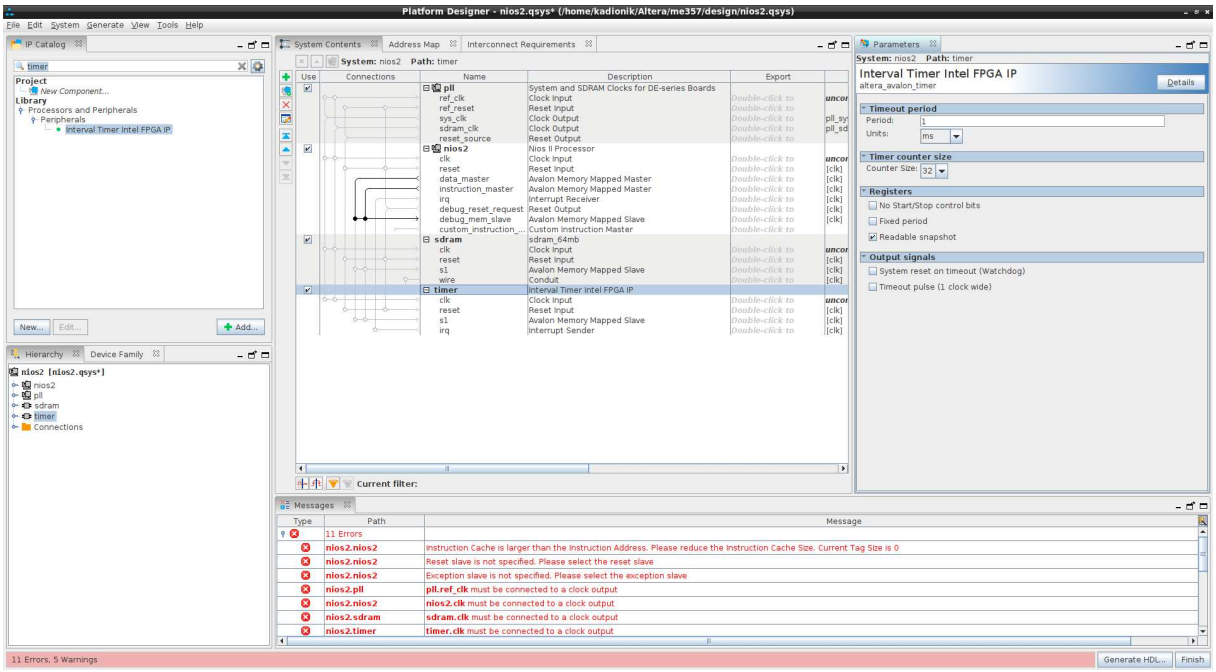
Bloc IP *nios2* (9)

- *Ajouter* le bloc IP *SDRAM*. Dans la fenêtre *IP Catalog*, rechercher les occurrences commençant par sdram. On choisira le bloc IP *sdram_64mb*. On ne touchera pas à la valeur par défaut des autres champs. On renommera le bloc IP *sdram*. On obtient la figure suivante :



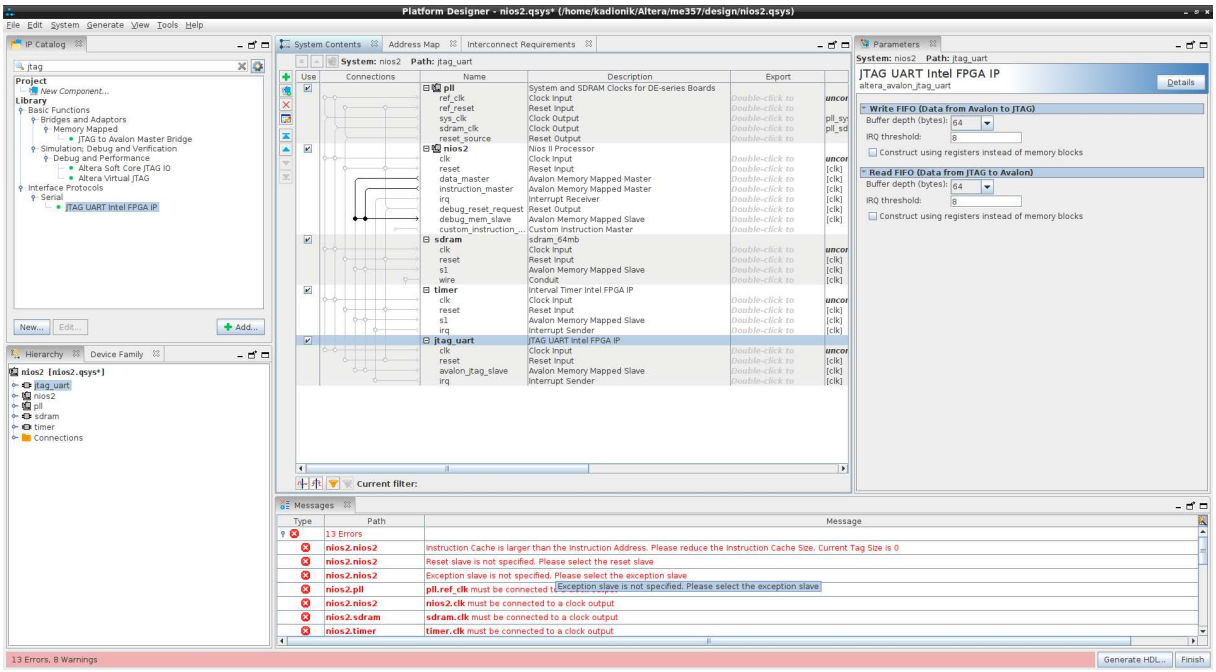
Bloc IP *sdram* (10)

- *Ajouter* le bloc IP *Timer*. Dans la fenêtre *IP Catalog*, rechercher les occurrences commençant par timer. On choisira le bloc *IP Interval Timer Intel FPGA IP*. On ne touchera pas à la valeur par défaut des autres champs. On renommera le bloc IP *timer*. On obtient la figure suivante :



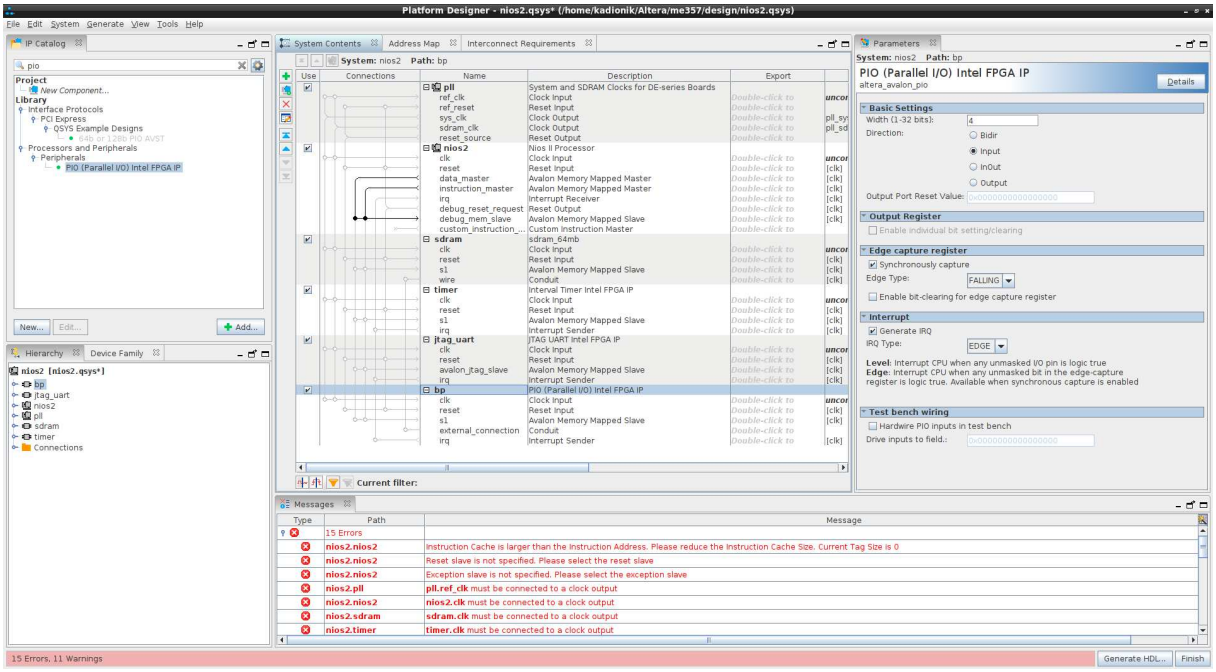
Bloc IP *timer* (11)

- Ajouter le bloc IP *JTAG/UART* (port série par l'interface JTAG). Dans la fenêtre *IP Catalog*, rechercher les occurrences commençant par jtag. On choisira le bloc IP *JTAG UART Intel FPGA IP*. On ne touchera pas à la valeur par défaut des autres champs. On renommera le bloc IP *jtag_uart*. On obtient la figure suivante :



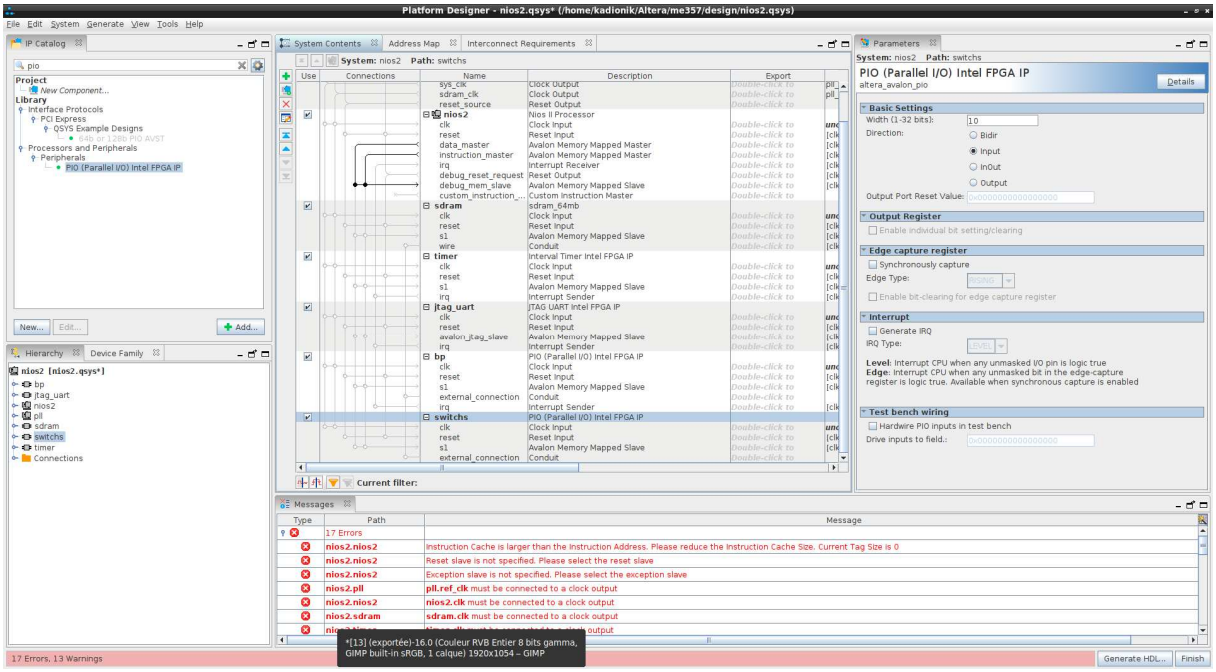
Bloc IP *jtag_uart* (12)

- Ajouter le bloc IP *PIO* (port parallèle). Dans la fenêtre *IP Catalog*, rechercher les occurrences commençant par pio. On choisira le bloc IP *PIO (Parallel I/O) Intel FPGA IP*. On configurera le bloc IP en entrée (*input*), avec une largeur de 4 bits, avec un registre de capture sur front descendant et capture synchrone (*edge capture register*, *synchronously capture*) et avec génération d'interruptions sur front (*generate IRQ*, *IRQ type edge*). On ne touchera pas à la valeur par défaut des autres champs. On renommera le bloc IP *bp*. On obtient la figure suivante :



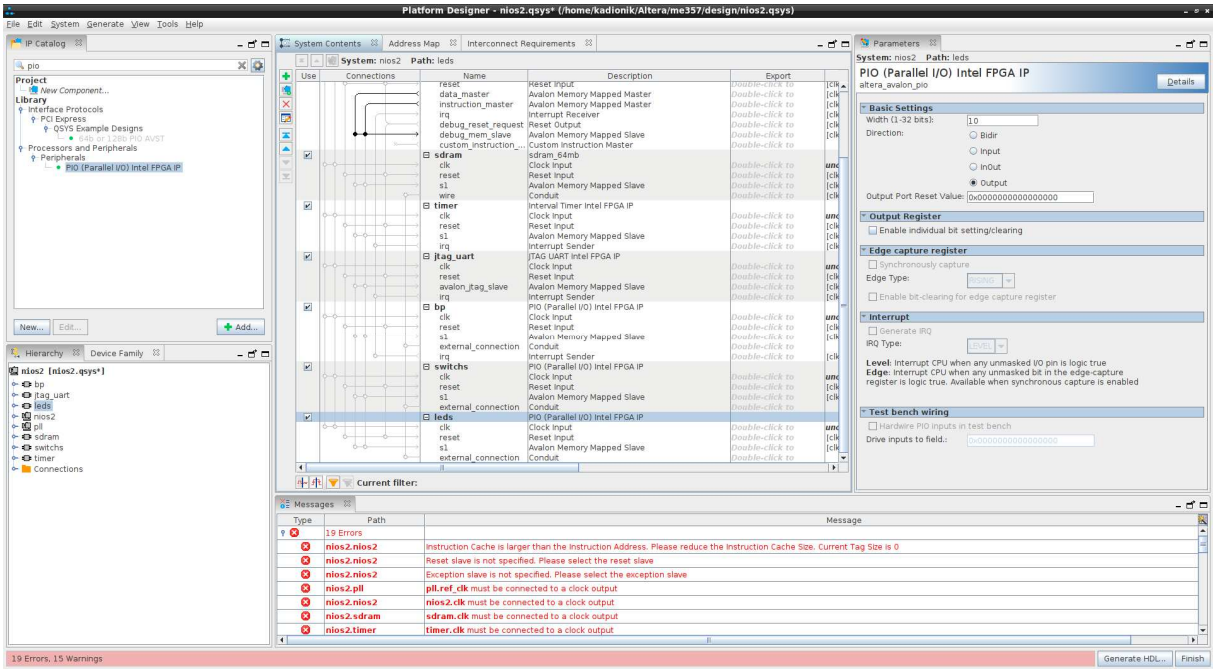
Bloc IP *bp* (13)

- Ajouter le bloc IP *PIO*. Dans la fenêtre *IP Catalog*, rechercher les occurrences commençant par pio. On choisira le bloc IP *PIO (Parallel I/O) Intel FPGA IP*. On configurera le bloc IP en entrée (*input*), avec une largeur de 10 bits. On ne touchera pas à la valeur par défaut des autres champs. On renommera le bloc IP *switchs*. On obtient la figure suivante :



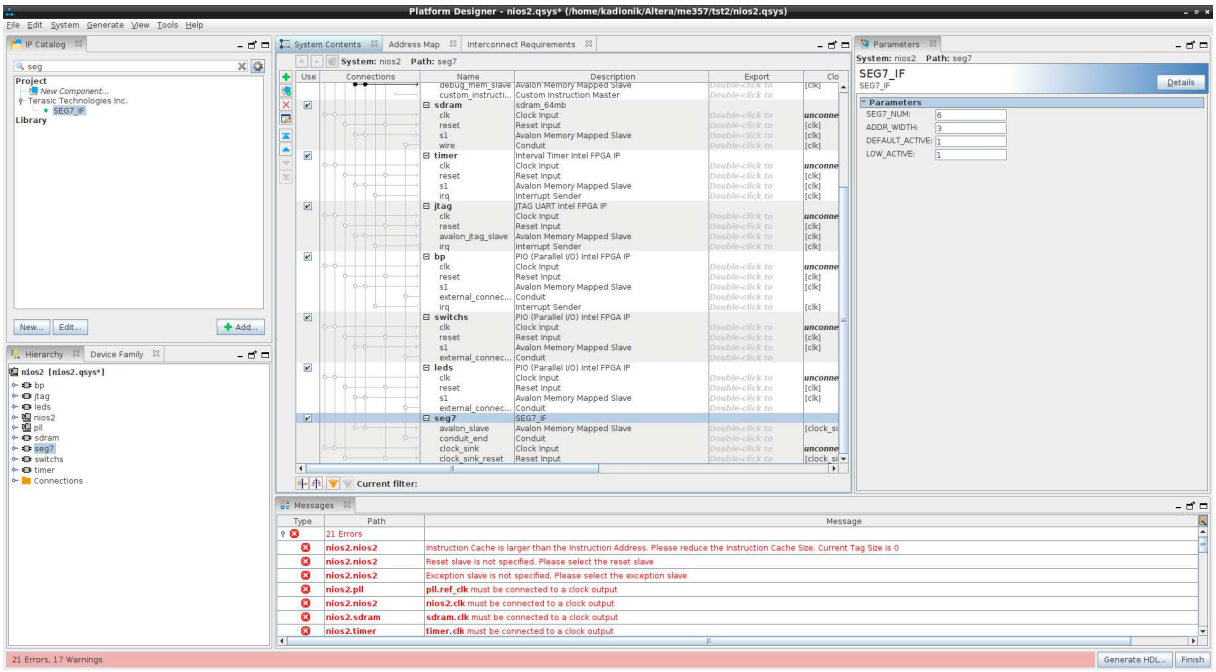
Bloc IP *switchs* (14)

- Ajouter le bloc IP *PIO*. Dans la fenêtre *IP Catalog*, rechercher les occurrences commençant par pio. On choisira le bloc IP *PIO (Parallel I/O) Intel FPGA IP*. On configurera le bloc IP en sortie (*output*), avec une largeur de 10 bits. On ne touchera pas à la valeur par défaut des autres champs. On renommera le bloc IP *leds*. On obtient la figure suivante :



Bloc IP *leds* (15)

- Ajouter enfin le bloc IP *7SEG_IF* (7 segments). Dans la fenêtre *IP Catalog*, rechercher les occurrences commençant par seg. On choisira le bloc IP *7SEG_IF*. On configurera le bloc IP avec *SEG7_NUM* égal à 6. On ne touchera pas à la valeur par défaut des autres champs. On renommera le bloc IP *seg7*. On obtient la figure suivante :

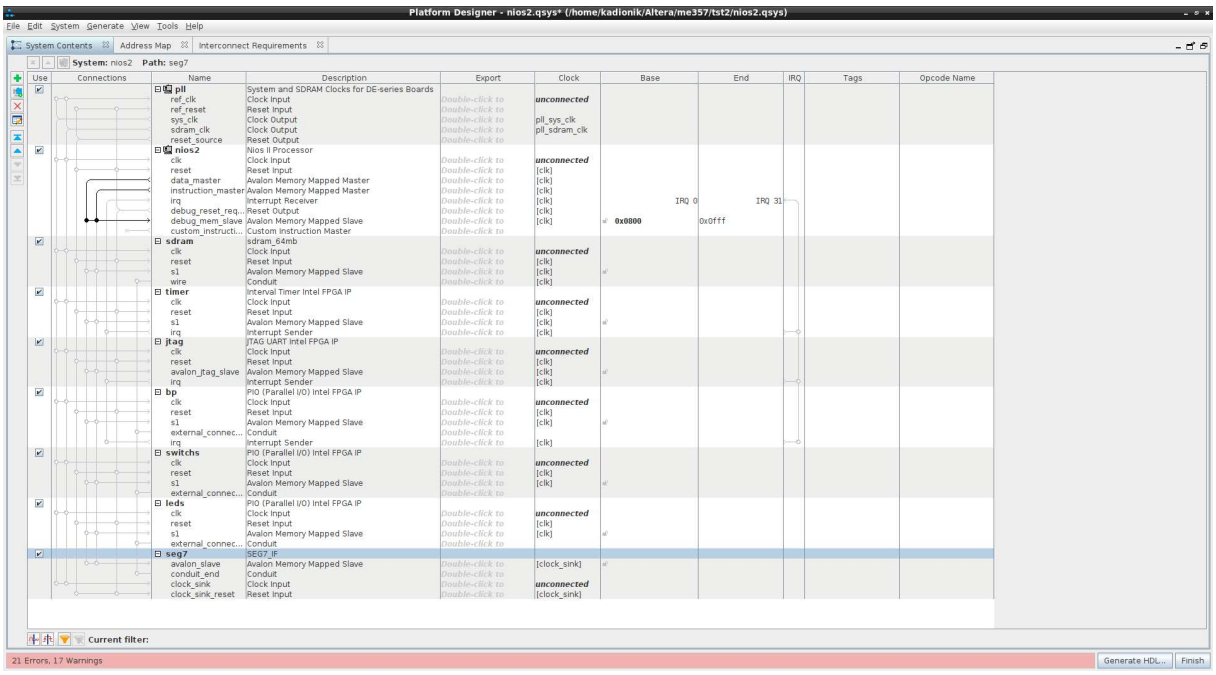


Bloc IP *seg7* (16)

Nous avons intégré tous les blocs IP dont nous avons besoin dans notre système SoPC.

Bien sûr, le travail n'est pas terminé car nous avons encore des erreurs à corriger en ce qui concerne la connexion de l'horloge, du reset, des bus de données et d'instructions, des interruptions, de l'initialisation de la table des vecteurs d'interruption et du lien avec le fichier Verilog `DE10_Standard_golden_top.v`.

Si tout va bien, on obtient la figure suivante :



Système SoPC *DE10_Standard_golden_top* (17)

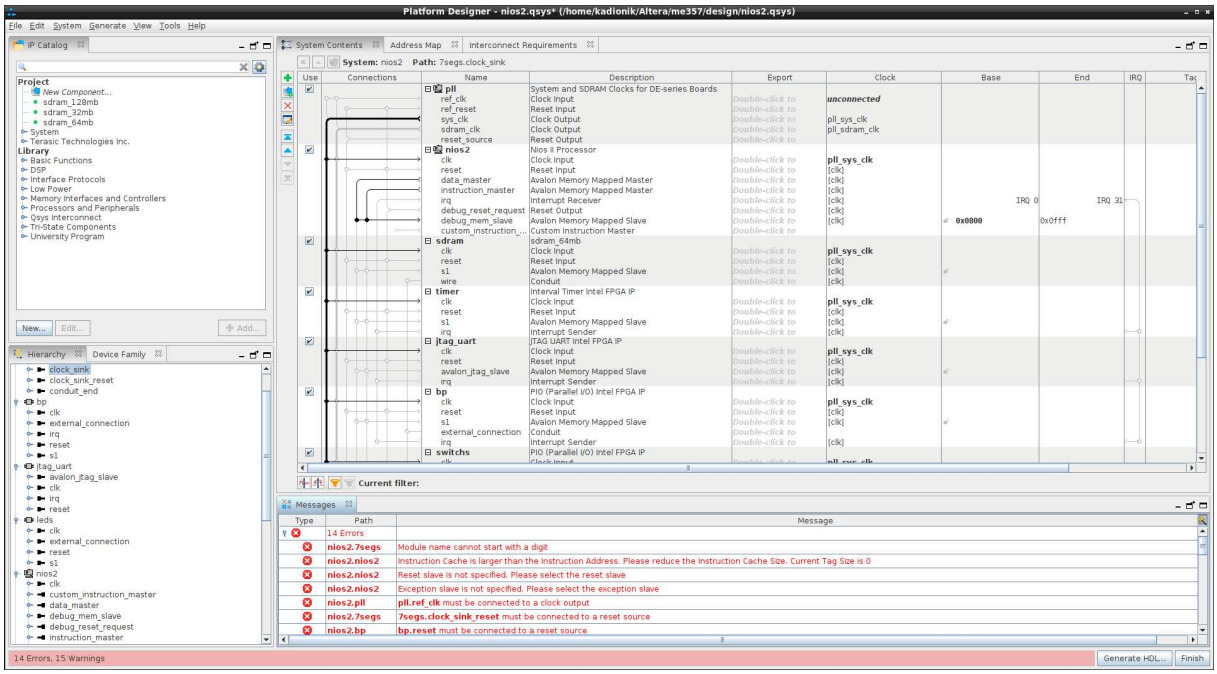
Nous en avons fini avec l'addition de blocs IP dans le système SoPC.

3.3. Ajout des connexions de signaux

Nous devons connecter les signaux d'horloge, de reset, de bus de données à chaque périphérique mais aussi connecter le bus d'instructions à la mémoire SDRAM.

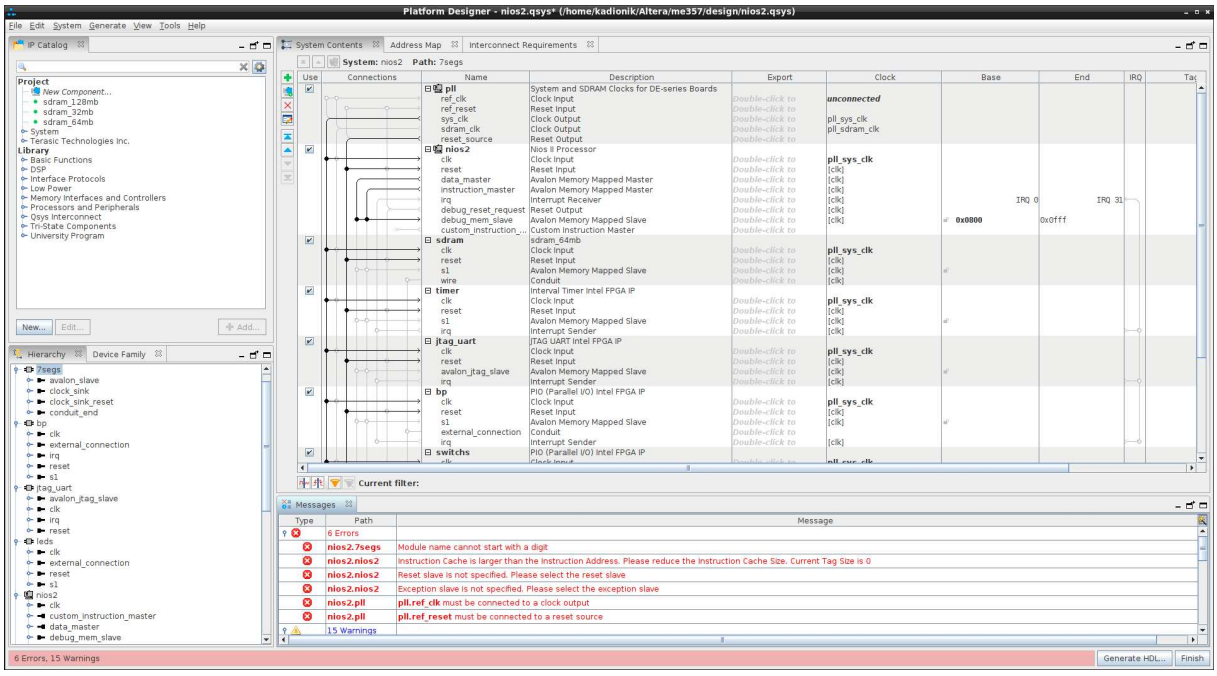
Dans la fenêtre *Platform Designer*, il y a dans la colonne *Connections* un réseau (X, Y) de connexions de signaux. Il suffit de cliquer à l'interconnexion de 2 signaux pour les interconnecter.

- Interconnecter le signal sortant *sys_clk* du bloc IP *pll* à l'entrée *clk* des autres blocs IP *nios2*, *sdram*, *timer*, *jtag_uart*, *bp*, *switchs*, *leds* et *seg7* (*clock_sink* pour le bloc IP *seg7*). On obtient la figure suivante :



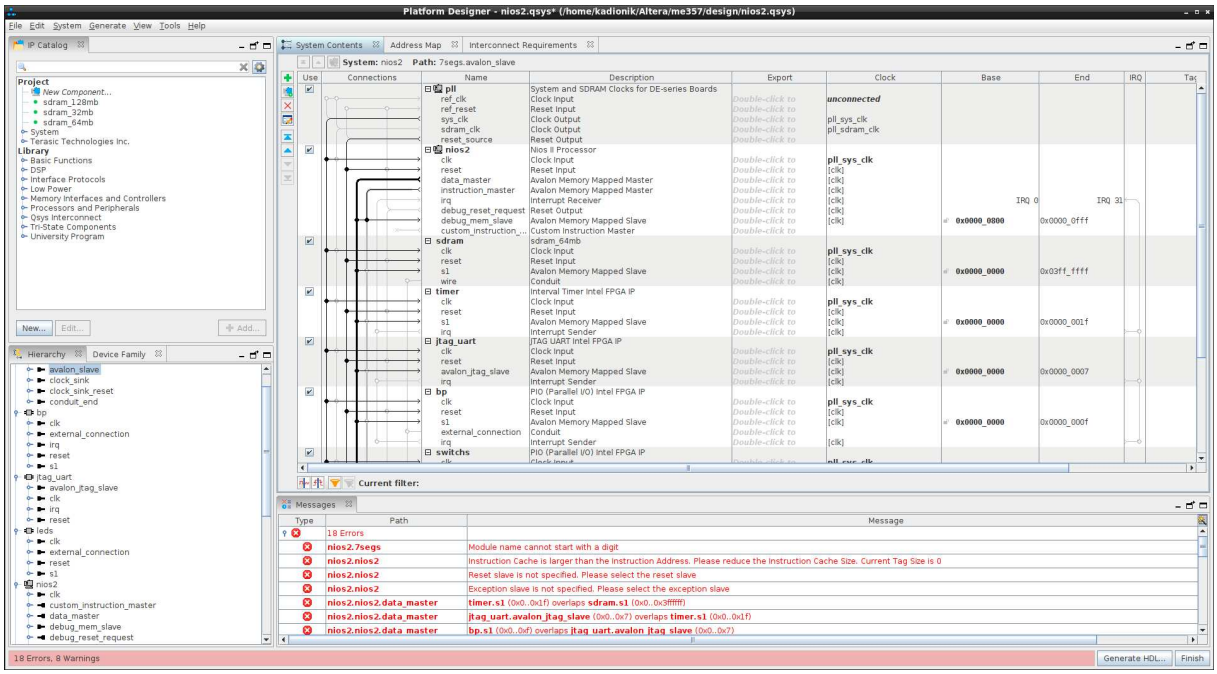
Connexion du signal d'horloge *sys_clk* (18)

- Interconnecter le signal sortant *reset_source* du bloc IP *pll* à l'entrée *reset* des autres blocs IP *nios2*, *sdram*, *timer*, *jtag_uart*, *bp*, *switchs*, *leds* et *seg7* (*clock_sink_reset* pour le bloc IP *seg7*) . On obtient la figure suivante :



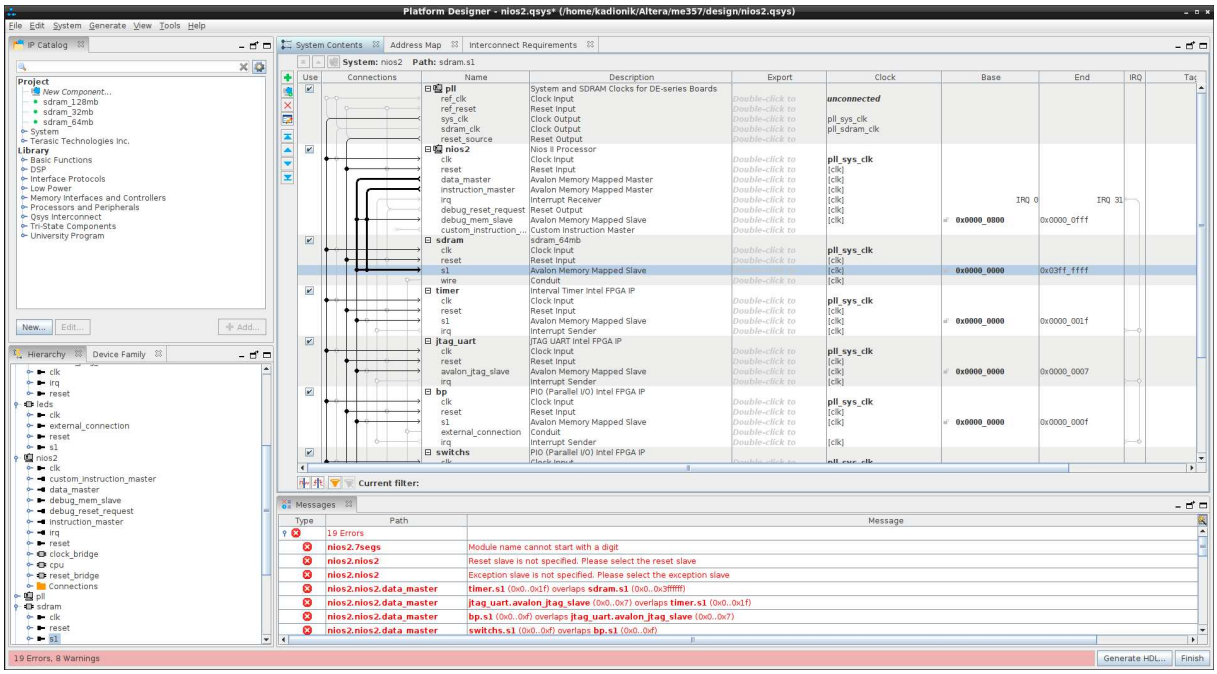
Connexion du signal de reset *reset_source* (19)

- Interconnecter le signal sortant *data_master* du bloc IP nios2 à l'entrée *s1* des autres blocs IP *sdram*, *timer*, *jtag_uart (avalon_jtag_slave* pour le bloc IP *jtag_uart*), *bp*, *switchs*, *leds* et *seg7* (*avalon_slave* pour le bloc IP *seg7*) . On obtient la figure suivante :



Connexion du signal *data_master* (20)

- Interconnecter le signal sortant *instruction_master* du bloc IP *nios2* à l'entrée *s1* du bloc IP *sdram* uniquement. En effet, la mémoire SDRAM contient les données et les instructions d'un programme à exécuter. On obtient la figure suivante :



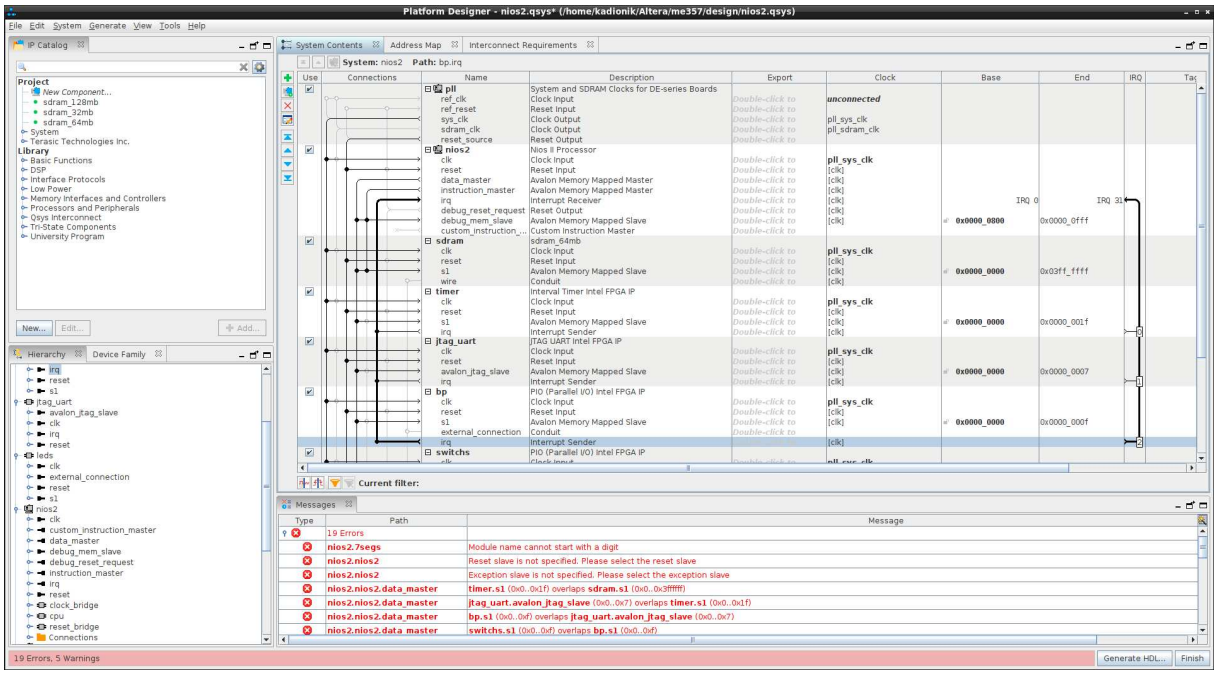
Connexion du signal *instruction_master* (21)

Nous en avons fini avec les interconnexions de signaux du système SoPC.

3.4. Ajout des interruptions

Les blocs IP *timer*, *jtag_uart* et *bp* peuvent travailler sous interruption. On doit donc préciser leur numéro d'interruption, de 0 à 31.

- Dans la fenêtre *Platform Designer*, il y a dans la colonne IRQ le réseau de connexions des interruptions. Pour ces 3 périphériques, il suffit de cliquer sur le petit losange pour définir le numéro d'interruption. Peut importe la valeur. On obtient la figure suivante (avec pour les valeurs d'IRQ : 0 à 2 ici) :



Ajout des interruptions (22)

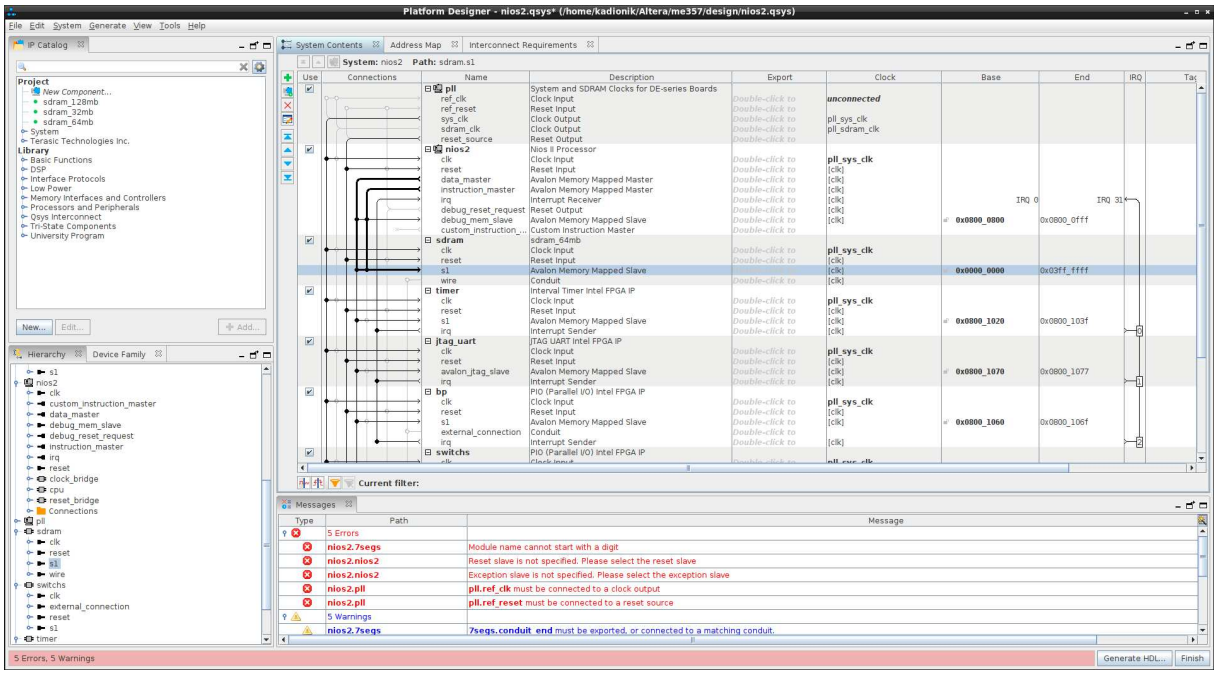
Nous en avons fini avec les interruptions du système SoPC.

3.5. Définition du mapping mémoire

Il faut définir le mapping mémoire des périphériques du système SoPC pour qu'il n'y ait pas de recouvrement mémoire.

- Choisir le menu *System > Assign Base Addresses*. On observe que le nombre de messages d'erreurs en rouge a drastiquement diminué.

- Mapper la mémoire SDRAM à partir de l'adresse 0x0000_0000. Pour cela, on cliquera dans la colonne *Base* de la fenêtre *Platform Designer* sur la valeur qui a été attribuée par défaut (0x0400_000) pour l'ajuster à 0x0000_0000. On obtient la figure suivante :



Mapping mémoire (23)

Nous en avons fini avec le mapping mémoire du système SoPC.

3.6. Définition des vecteurs d'exception

Le processeur NIOS II a 2 vecteurs d'exception :

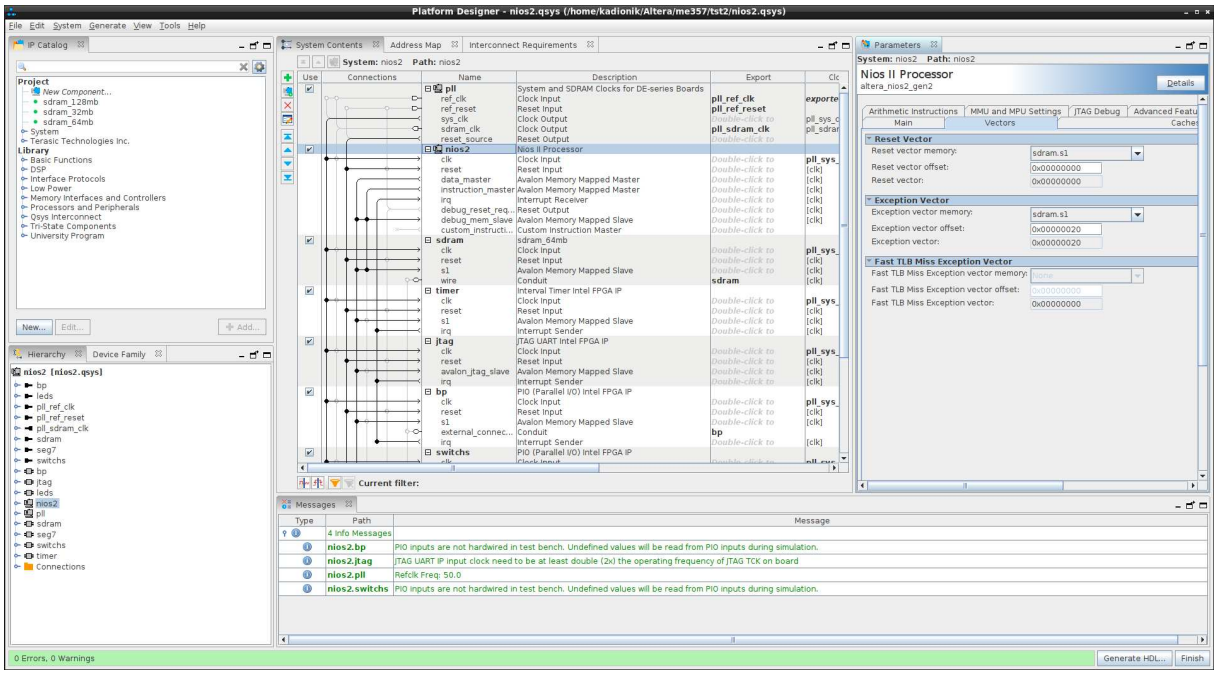
- le vecteur de reset qui précise le point d'entrée du programme exécuté au reset du processeur.
- le vecteur d'interruption en cas d'occurrence d'une interruption. Il précise le point d'entrée de la routine d'interruption ISR (*Interrupt Sub Routine*). Cette routine est commune aux 32 sources d'interruption possibles et il faudra donc dans la routine d'interruption rechercher qui est la cause de l'interruption.

- Dans la fenêtre *Platform Designer*, on double clique sur le bloc IP *nios2*. Dans la fenêtre *Parameters* qui apparait à droite, on choisit l'onglet *Vectors*.

- On choisit pour le champ *Reset vector memory* la valeur *sdram.s1*.

- On choisit pour le champ *Exception vector memory* la valeur *sdram.s1*.

Les deux vecteurs d'exception sont installés dans la mémoire SDRAM aux adresses respectives 0x0000_0000 et 0x0000_0020. On laisse les autres valeurs par défaut. On obtient la figure suivante :



Vecteurs d'exception (24)

Nous en avons fini avec les vecteurs d'exception du système SoPC.

3.7. Exportation des signaux externes au circuit FPGA

Certains signaux du système SoPC sont externes au circuit FPGA car ils sont connectés aux périphériques externes comme la mémoire SDRAM, les boutons poussoir, les switchs, les leds et les afficheurs 7 segments. Il y a aussi l'horloge externe et le reset.

Dans la fenêtre *Platform Designer*, il y a la colonne *Export* pour l'exportation des signaux externes au circuit FPGA.

- Dans la fenêtre *Platform Designer*, on double clique sur le champ *Export* du signal *ref_clk* du bloc IP *pll*. Il apparait la valeur *pll_ref_clk* que l'on gardera par défaut.

- On double clique sur le champ *Export* du signal *ref_reset* du bloc IP *pll*. Il apparait la valeur *pll_ref_reset* que l'on gardera par défaut.

- On double clique sur le champ *Export* du signal *sdram_clk* du bloc IP *pll*. Il apparait la valeur *pll_sdram_clk* que l'on gardera par défaut.

- On double clique sur le champ *Export* du signal *wire* (*Conduit*) du bloc IP *sdram*. Il apparait la valeur *sdram_wire* que l'on changera en *sdram*.

- On double clique sur le champ *Export* du signal *external_connection* (*Conduit*) du bloc IP *bp*. Il apparait la valeur *bp_external_connection* que l'on changera en *bp*.

- On double clique sur le champ *Export* du signal *external_connection* (*Conduit*) du bloc IP *switchs*. Il apparait la valeur *switchs_external_connection* que l'on changera en *switchs*.

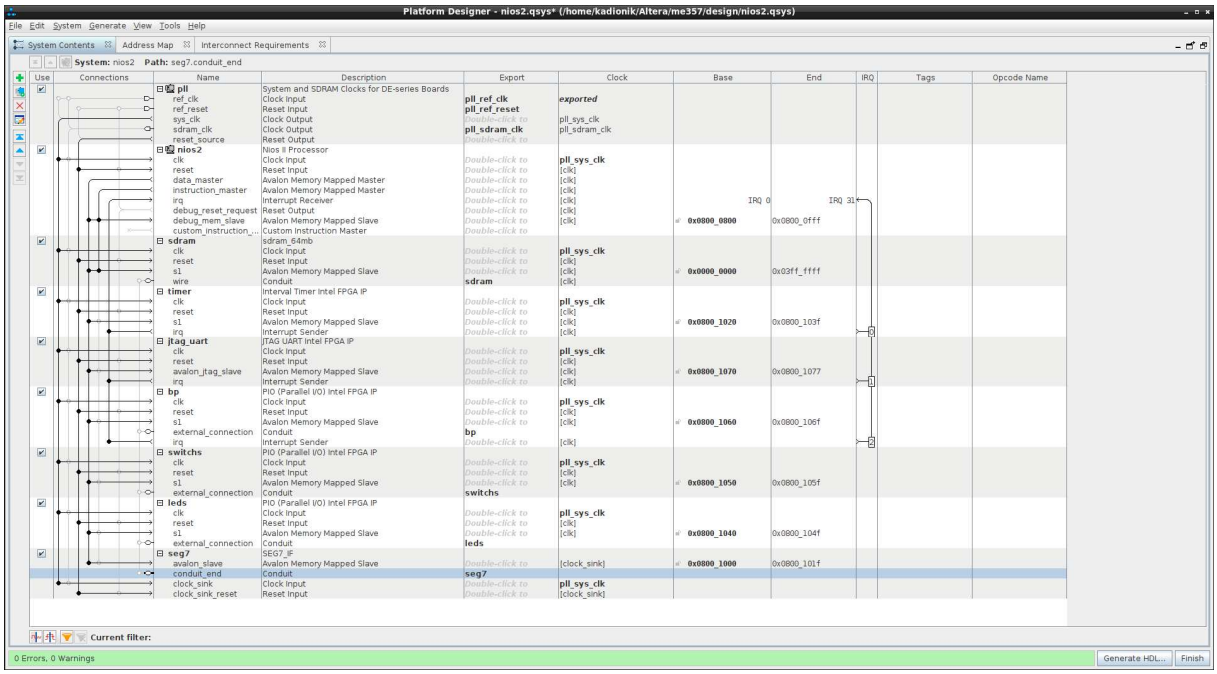
- On double clique sur le champ *Export* du signal *external_connection* (*Conduit*) du bloc IP *leds*. Il apparait la valeur *leds_external_connection* que l'on changera en *leds*.

- Enfin, on double clique sur le champ *Export* du signal *conduit_end* (*Conduit*) du bloc IP *seg7*. Il apparait la valeur *seg7_conduit_end* que l'on changera en *seg7*.

On notera qu'il n'y a plus de messages d'erreurs.

Nous en avons fini avec l'exportation des signaux externes du système SoPC.

On obtient alors le système SoPC suivant :

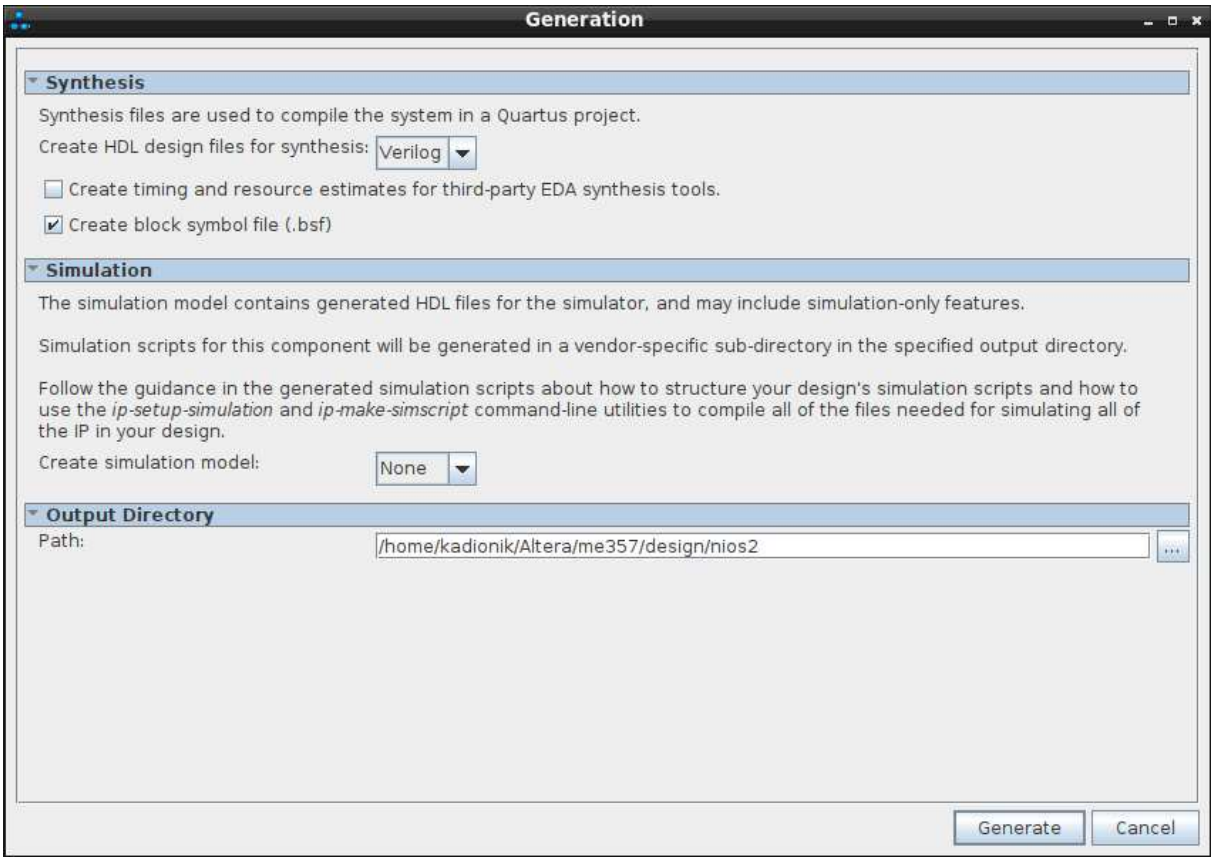


Système SoPC *DE10_Standard_golden_top* (25)

3.8. Génération HDL du système SoPC

Dans la fenêtre *Platform Designer*, on clique sur le bouton *Generate HDL* pour générer les sources HDL du système SoPC.

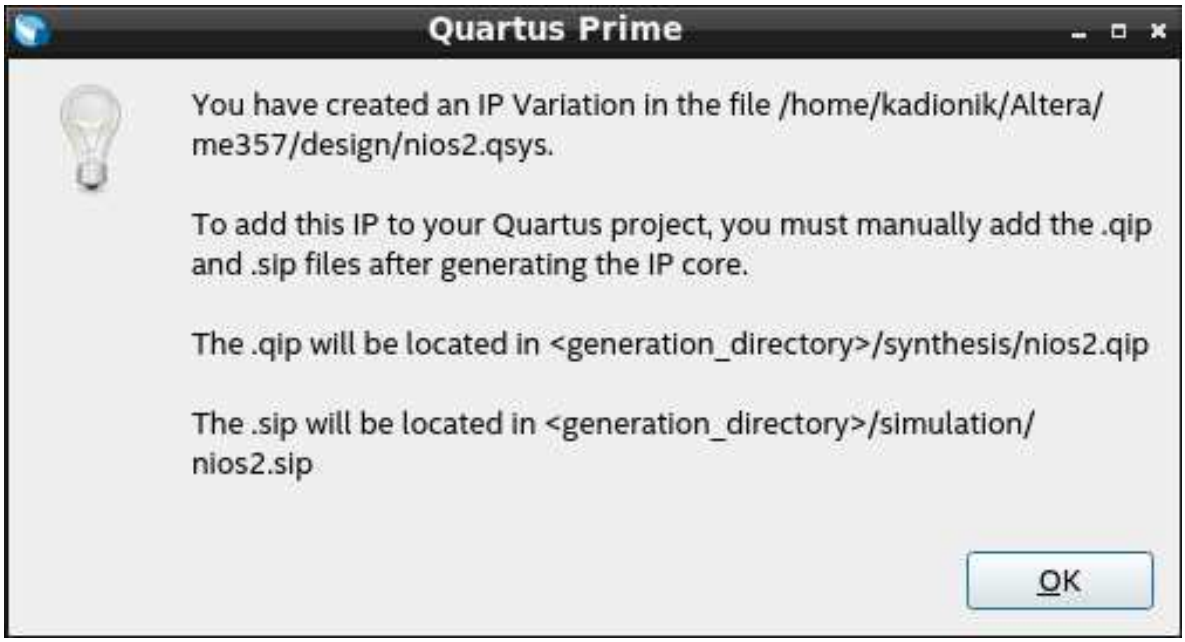
On obtient la fenêtre suivante :



Génération HDL du système SoPC *DE10_Standard_golden_top* (26)

- Cliquer sur le bouton *Generate*.
- Après génération, cliquer sur le bouton *Finish* pour sortir de *Platform Designer*.

La fenêtre d'indications suivante apparait :



Indication sur l'inclusion du fichier .qip (27)

Elle signale qu'il faudra intégrer le fichier `.qip` dans son projet *Quartus Prime* pour intégrer le système SoPC dans la synthèse.

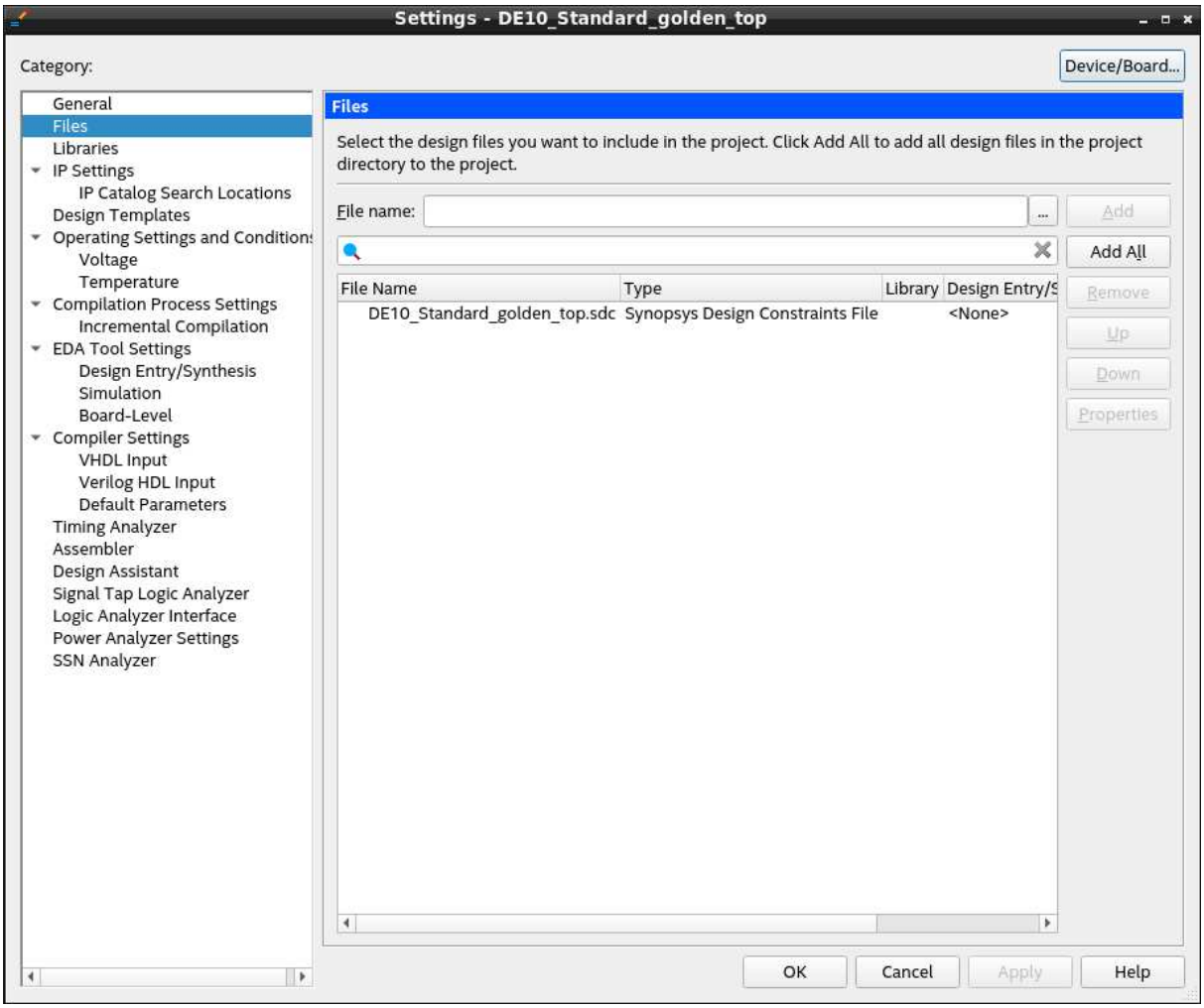
Notre système SoPC est sauvegardé dans le fichier `nios2.qsys`.

3.9. Synthèse du système SoPC

On revient maintenant à l'outil *Quartus Prime*.

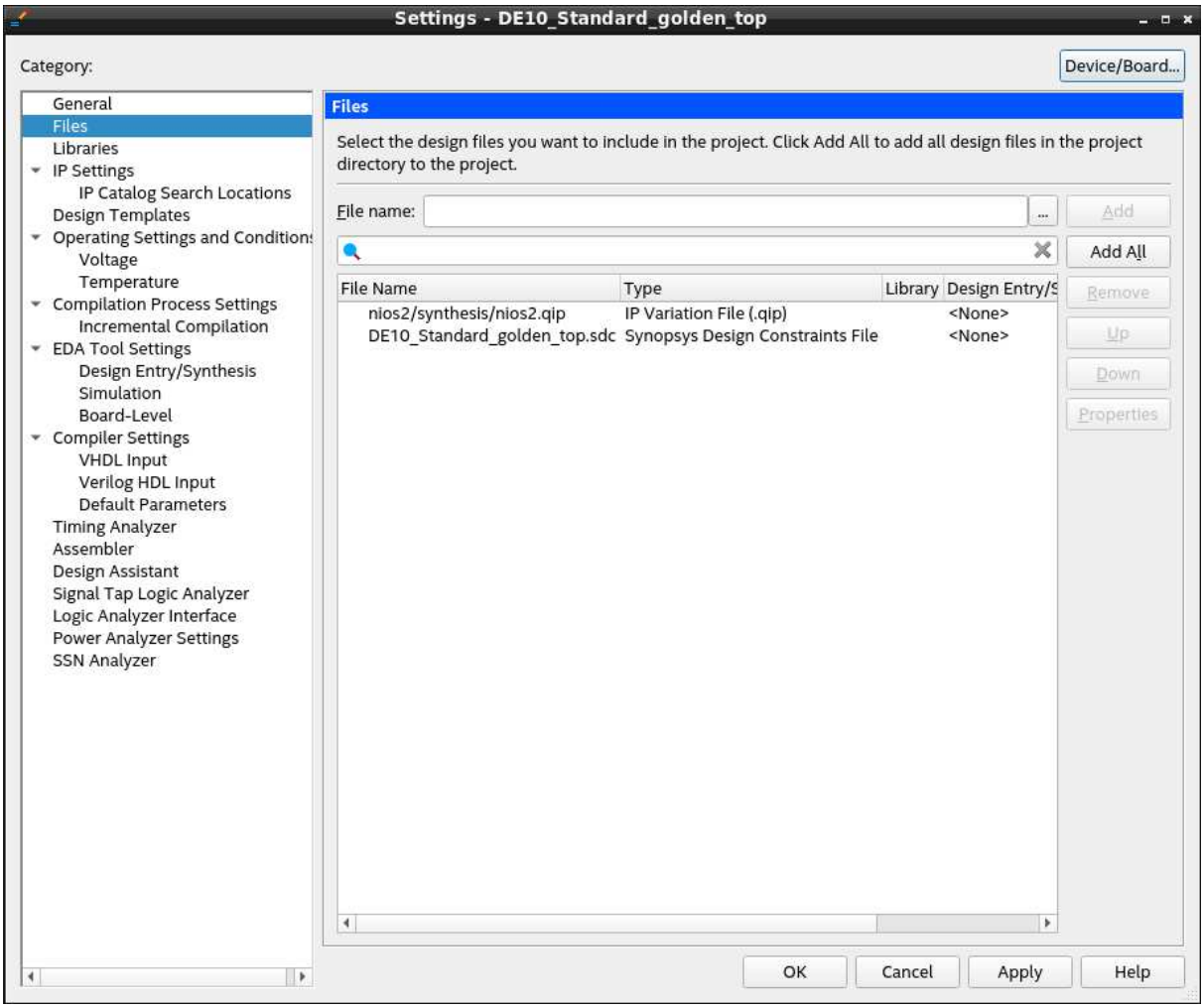
Le projet *DE10_Standard_golden_top* est toujours ouvert.

- Ajouter le fichier `nios2.qip` qui est sous `standard_nios/nios2/synthesis/` au projet *Quartus Prime*. On choisit le menu *Project > Add/Remove Files in Project*. On obtient la fenêtre suivante :



Ajout du fichier nios2.qip (28)

- Ajouter le fichier `nios2.qip` en cliquant sur le bouton "...". On obtient la fenêtre suivante :



Fichier nios2.qip ajouté (29)

Il faut enfin aussi modifier le fichier Verilog *Top-Level* `DE10_Standard_golden_top.v` pour faire le lien entre les signaux exportés (*Conduit*) du système SoPC et les signaux externes du niveau *Top-Level*.

L'outil *Platform Designer* a généré un canevas Verilog pour nous faciliter la tâche.

Sous *Quartus Prime*, on ouvrira le fichier *Top-Level* `DE10_Standard_golden_top.v` mais aussi le fichier canevas sous `standard_nios/nios2/nios2_inst.v`.

Le fichier `nios2_inst.v` contient :

```
nios2 u0 (
.pll_ref_clk_clk     (<connected-to-pll_ref_clk_clk>),     // pll_ref_clk.clk
.pll_ref_reset_reset (<connected-to-pll_ref_reset_reset>), // pll_ref_reset.reset
.pll_sdram_clk_clk   (<connected-to-pll_sdram_clk_clk>),   // pll_sdram_clk.clk
.sdram_ba            (<connected-to-sdram_ba>),            // sdram.ba
.sdram_addr          (<connected-to-sdram_addr>),          // .addr
.sdram_cas_n         (<connected-to-sdram_cas_n>),         // .cas_n
.sdram_cke           (<connected-to-sdram_cke>),           // .cke
.sdram_cs_n          (<connected-to-sdram_cs_n>),          // .cs_n
.sdram_dq            (<connected-to-sdram_dq>),            // .dq
.sdram_dqm           (<connected-to-sdram_dqm>),           // .dqm
.sdram_ras_n         (<connected-to-sdram_ras_n>),         // .ras_n
.sdram_we_n          (<connected-to-sdram_we_n>),          // .we_n
.bp_export           (<connected-to-bp_export>),           // bp.export
.switchs_export      (<connected-to-switchs_export>),      // switchs.export
.leds_export         (<connected-to-leds_export>),         // leds.export
.seg7_export         (<connected-to-seg7_export>)          // seg7.export
);
```

Il faudra copier ce canevas dans le fichier `DE10_Standard_golden_top.v` qu'il faudra ensuite modifier pour raccrocher les signaux externes du système SoPC aux signaux externes du circuit FPGA.

Le fichier `DE10_Standard_golden_top.v` devient alors :

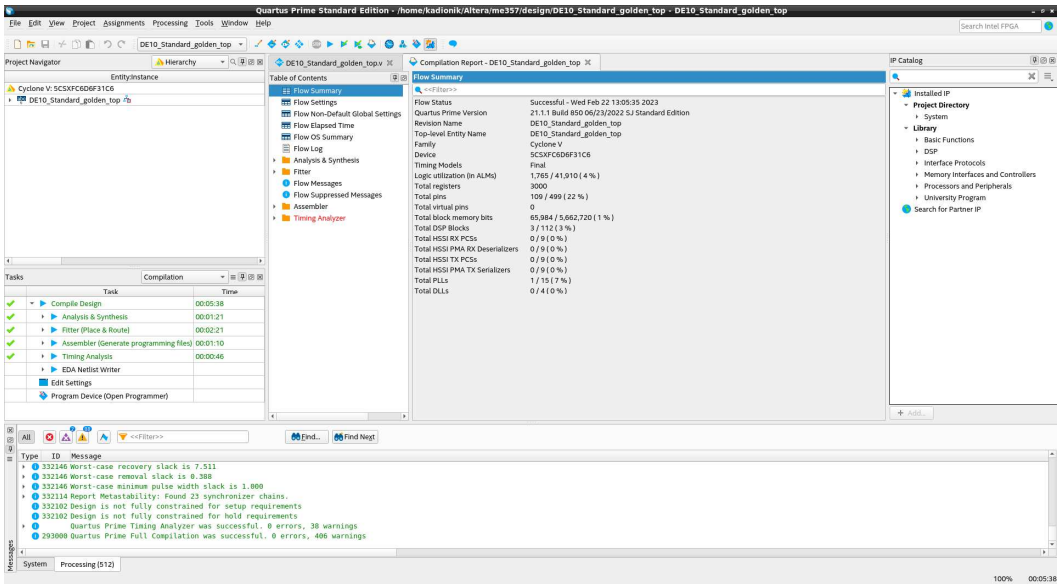
```
module DE10_Standard_golden_top(

      ///////// CLOCK /////////
      input              CLOCK2_50,
      input              CLOCK3_50,
      input              CLOCK4_50,
      input              CLOCK_50,

      ///////// KEY /////////
      input    [ 3: 0]   KEY,

      ///////// SW /////////
      input    [ 9: 0]   SW,

      ///////// LED /////////
      output   [ 9: 0]   LEDR,

      ///////// Seg7 /////////
      output   [ 6: 0]   HEX0,
      output   [ 6: 0]   HEX1,
      output   [ 6: 0]   HEX2,
      output   [ 6: 0]   HEX3,
      output   [ 6: 0]   HEX4,
      output   [ 6: 0]   HEX5,

      ///////// SDRAM /////////
      output             DRAM_CLK,
      output             DRAM_CKE,
      output   [12: 0]   DRAM_ADDR,
      output   [ 1: 0]   DRAM_BA,
      inout    [15: 0]   DRAM_DQ,
      output             DRAM_LDQM,
      output             DRAM_UDQM,
      output             DRAM_CS_N,
      output             DRAM_WE_N,
      output             DRAM_CAS_N,
      output             DRAM_RAS_N
);


wire  HEX0P;
wire  HEX1P;
wire  HEX2P;
wire  HEX3P;
```

```
wire  HEX4P;
wire  HEX5P;

nios2 u0 (
.pll_ref_clk_clk           (CLOCK_50),
.pll_ref_reset_reset       (1'b0),

.pll_sdram_clk_clk         (DRAM_CLK),
.sdram_addr                (DRAM_ADDR),
.sdram_ba                  (DRAM_BA),
.sdram_cas_n               (DRAM_CAS_N),
.sdram_cke                 (DRAM_CKE),
.sdram_cs_n                (DRAM_CS_N),
.sdram_dq                  (DRAM_DQ),
.sdram_dqm                 ({DRAM_UDQM,DRAM_LDQM}),
.sdram_ras_n               (DRAM_RAS_N),
.sdram_we_n                (DRAM_WE_N),

.bp_export                 (KEY),
.switchs_export            (SW),
.leds_export               (LEDR),
.seg7_export    ({HEX5P,HEX5,HEX4P,HEX4,HEX3P,HEX3,HEX2P,HEX2,HEX1P,HEX1,HEX0P,HEX0})
);

endmodule
```

On note l'ajout des signaux supplémentaires `HEX0P` à `HEX5P` qui sont les anodes communes des 6 afficheurs 7 segments nécessaires au bloc IP *7SEG_IF*.
Il ne reste plus qu'à synthétiser le système SoPC pour générer le fichier de programmation du circuit FPGA `.sof`.

- Choisir le menu *Processing > Start Compilation*.

- Après synthèse, quelles sont les ressources consommées et combien dans le circuit FPGA ? Après synthèse, on obtient la figure suivante :

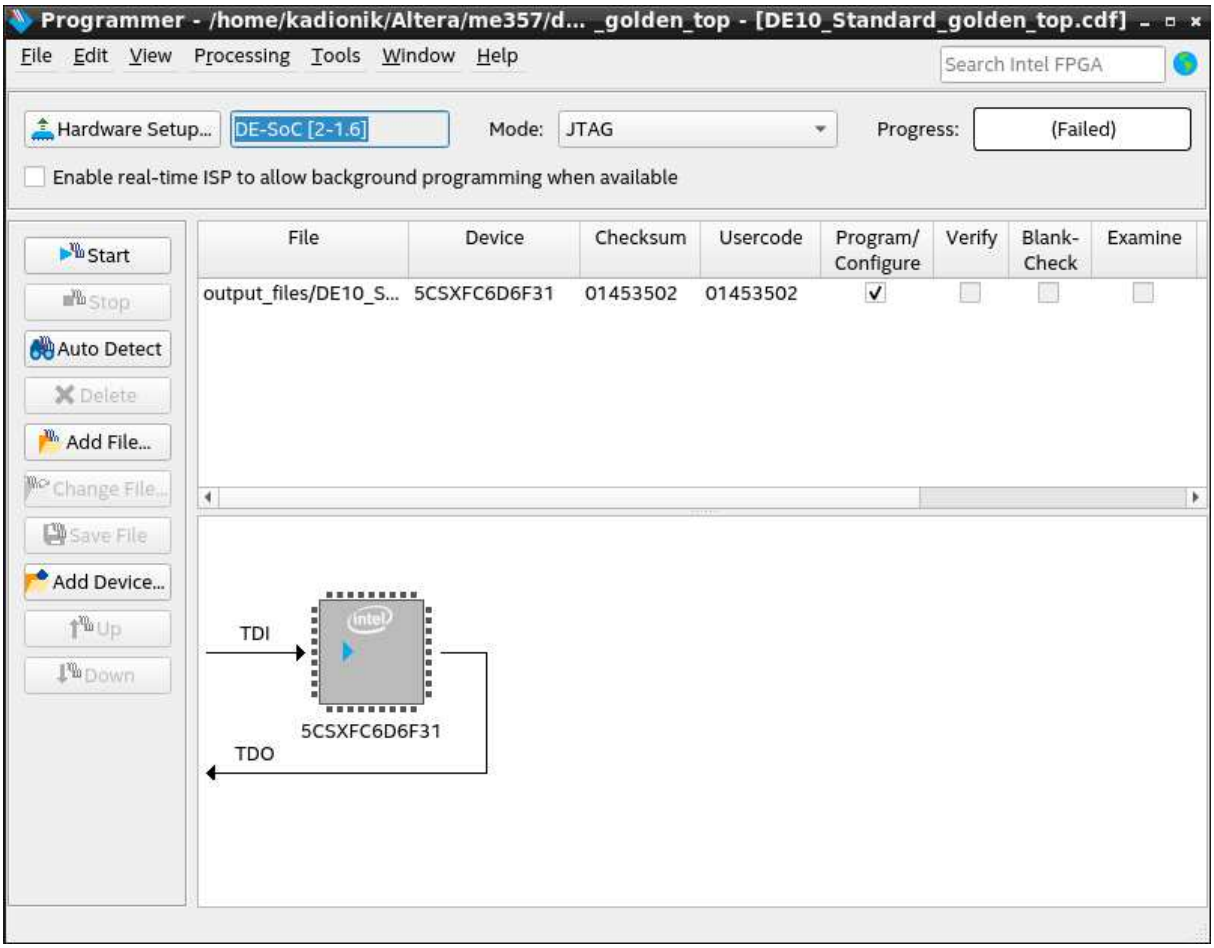


Synthèse du système SoPC (30)

3.10. Programmation du circuit FPGA

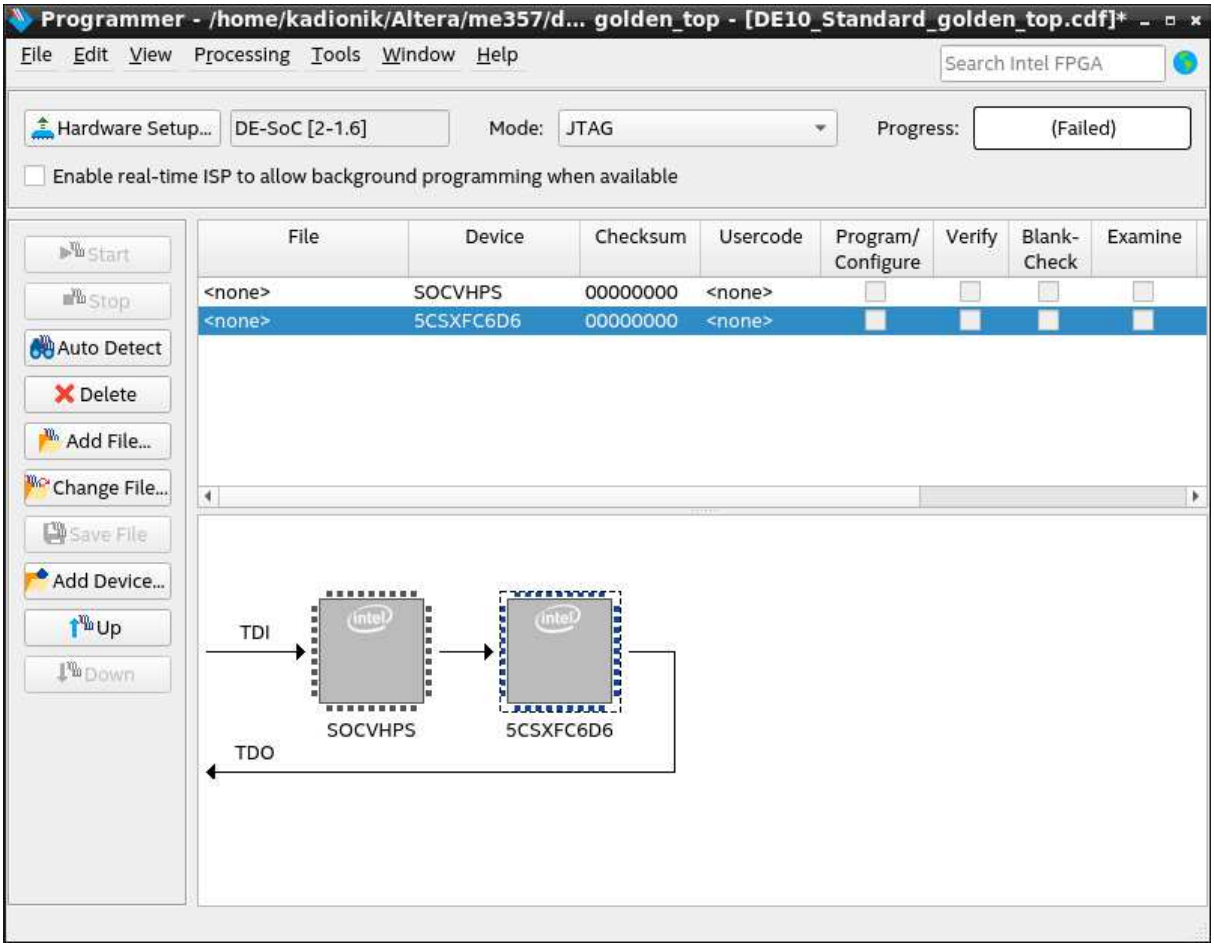
A partir de *Quartus Prime*, on peut programmer le circuit Cyclone V de la carte DE-Standard.

- Choisir le menu *Tool* > *Programmer*. La fenêtre suivante apparaît :



Fenêtre *Programmer* (31)

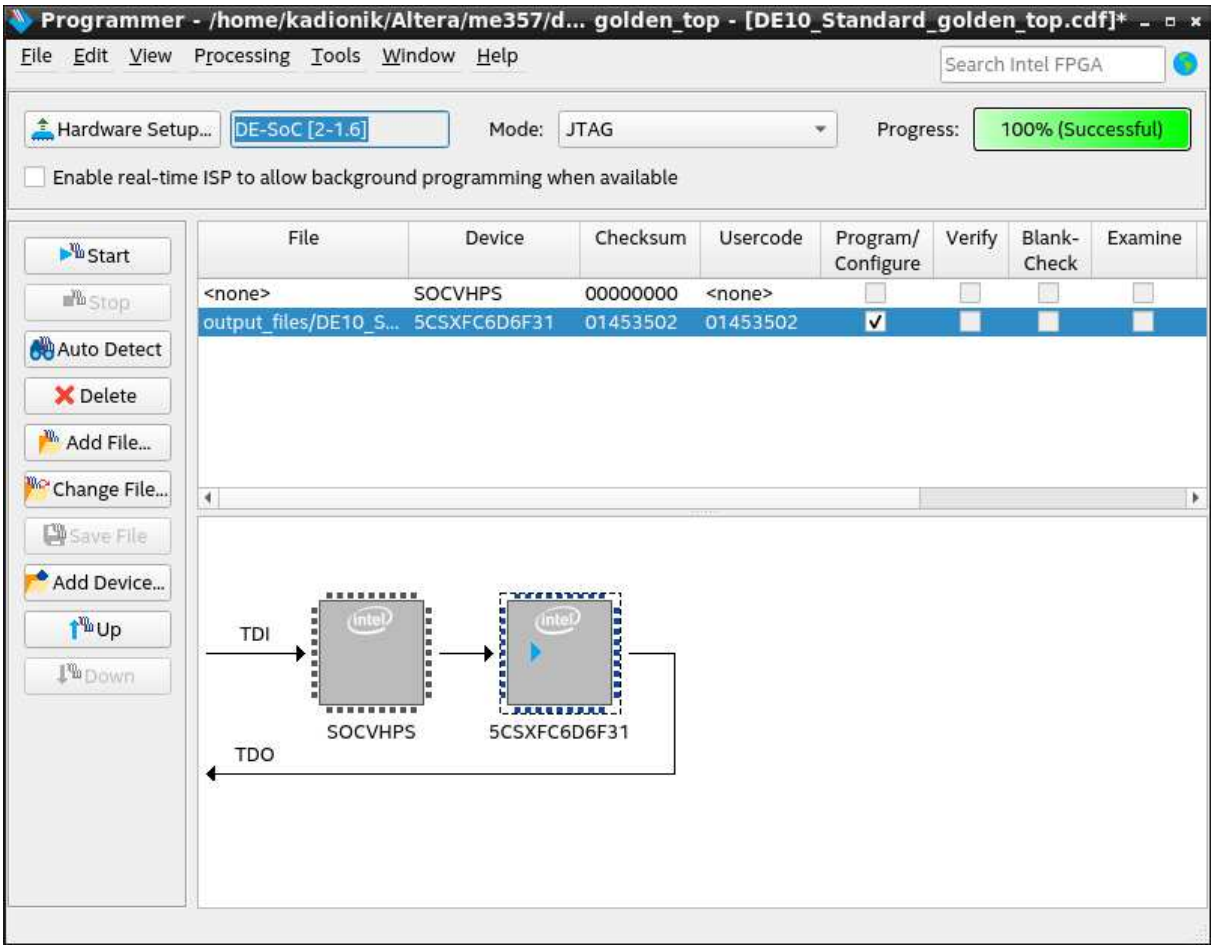
- Cliquer sur le bouton *Auto Detect* puis on choisira le *Device* 5CSXFC6D6. On obtient la figure suivante :



Sélection du circuit FPGA (32)

- Sélectionner ensuite le composant 5CSXFC6D6 puis on cliquera sur le bouton *Change File…* pour choisir le fichier `.sof`.

- Dans le répertoire `output_files/`, choisir le fichier `DE10_Standard_golden_top.sof`. On cochera la case *Program/Configure* pour ce composant puis on cliquera sur le bouton *Start* pour programmer le circuit FPGA. On obtient la figure suivante :



Programmation du circuit FPGA (33)

Nous en avons alors fini avec la partie matérielle de construction de notre système SoPC.

Il nous reste la deuxième partie du *codesign* qui correspond à la programmation logicielle.

On pourra toujours reprogrammer le circuit FPGA depuis *Eclipse*.

On peut enfin fermer *Quartus Prime*...

4. EX 2 INTEL : HELLO WORLD

Nous allons réaliser un premier test logiciel fonctionnel en implémentant sur la carte cible DE10-Standard le programme « *Hello World* ».

On utilisera par la suite *Eclipse* comme atelier de développement logiciel IDE.

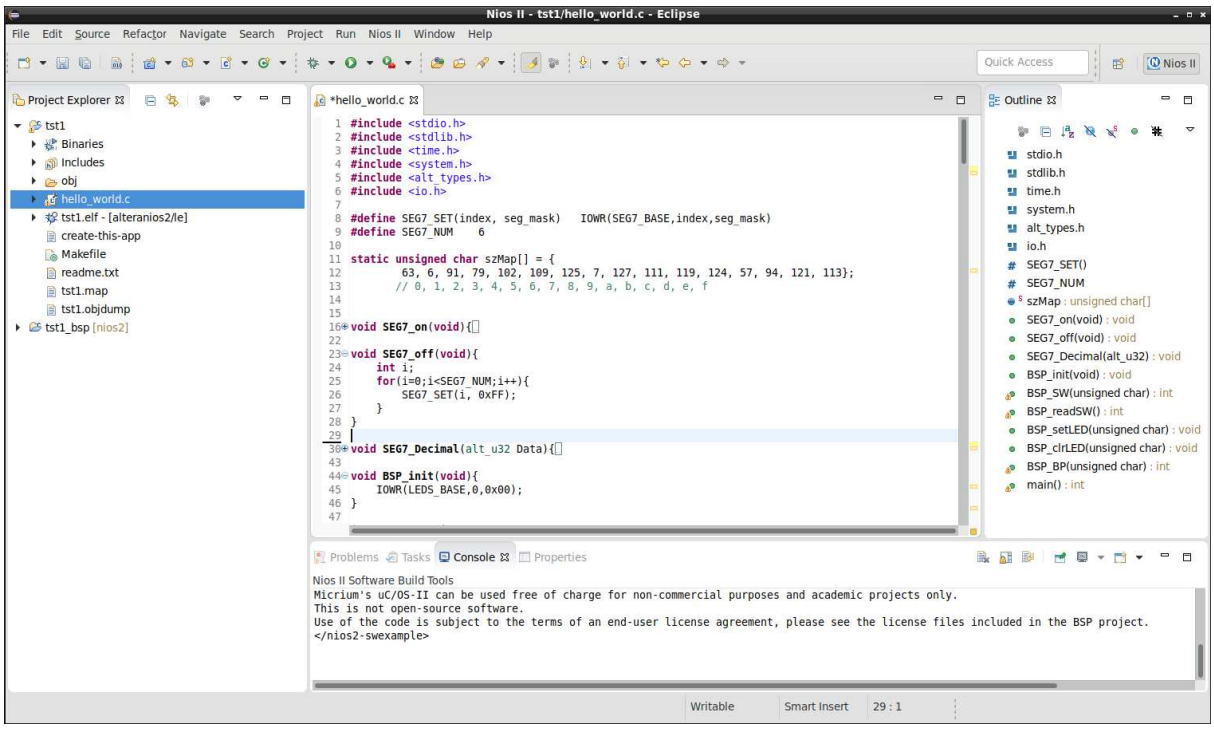
On utilisera pour cela le *shell script* `goeclipe` qui lancera *Eclipse* avec les bons paramètres.

Il faut aussi savoir qu'*Eclipse* utilise un espace de travail (*Workspace*) qui est généralement le répertoire `software/` que l'on a sous `de10/standard_nios/`.

- Se placer ensuite dans le répertoire `de10/standard_nios`.
  ```
  host% cd de10/standard_nios
  ```

- Se placer dans l'environnement de développement Intel si ce n'est pas déjà fait puis lancer *Eclipse*. On ajustera le *Workspace* à `de10/standard_nios/software/`. On a la fenêtre suivante :
  ```
  host% n2sdk
  [Xilinx EDK]$ goeclipse
  ```

L'allure d'*Eclipse* est donnée sur la figure suivante et a un fonctionnement similaire à l'IDE *Netbeans* :



IDE *Eclipse*

On vérifiera aussi que l'on a aussi dans le terminal de lancement d'*Eclipse* les traces suivantes :

```
Byte Stream Device: jtaguart_0
            Path:         /connections/DE-SoC         on         localhost         (2-
1.6)/5CSEBA6(.|ES)|5CSEMA6|..@2/(110:128 v1 #0)/jtaguart_0
Processor: nios2_0
            Path:         /connections/DE-SoC         on         localhost         (2-
1.6)/5CSEBA6(.|ES)|5CSEMA6|..@2/(70:34 v3 #0)/nios2
```

Cela signifie que l'on a bien accès au processeur NIOS II par le JTAG.

Si l'on n'a pas ces traces, il faudra programmer le circuit FPGA par le menu Eclipse *NIOS II > Quartus Prime Programmer*, sortir d'Eclipse et le relancer...

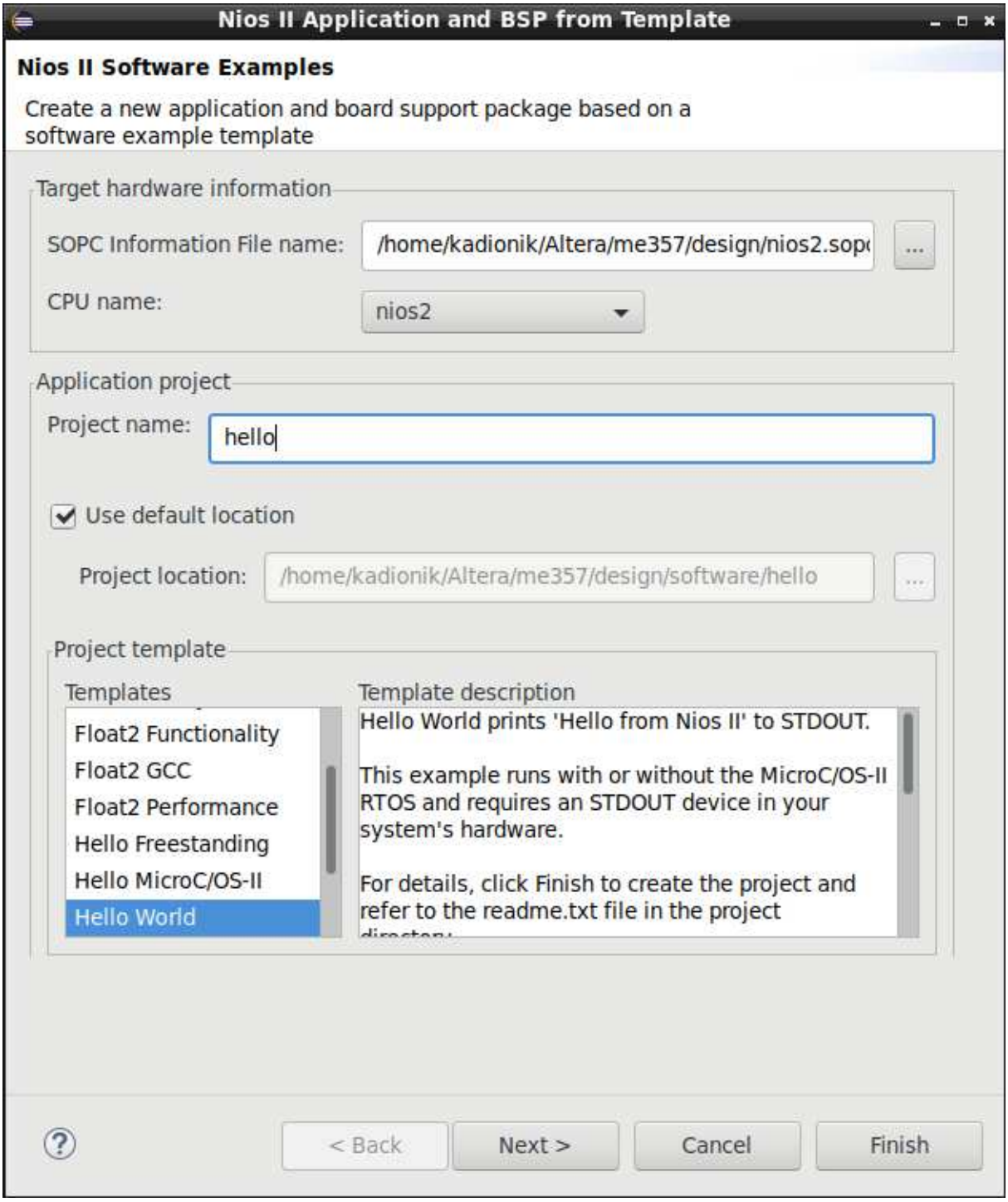
- Créer un nouveau projet *Eclipse hello* par le menu *File > New > NIOS II Application and BSP from Template*. On renseignera le champ *SOPC Information File name* avec le fichier `de10/standard_nios/nios2.sopcinfo`. Le fichier `.spocinfo` est un fichier verbeux de type XML qui donne toutes les informations sur le système SoPC et ses blocs IP.

- Choisir *hello* comme nom de projet pour le champ *Project name*.

- Choisir pour le *Project template* le *template Hello Word*.

- Cliquer sur le bouton *Next* puis sur le bouton *Finish*.

- A quoi correspond le projet *hello_bsp* ? On a la figure suivante :



Projet *Eclipse Hello*

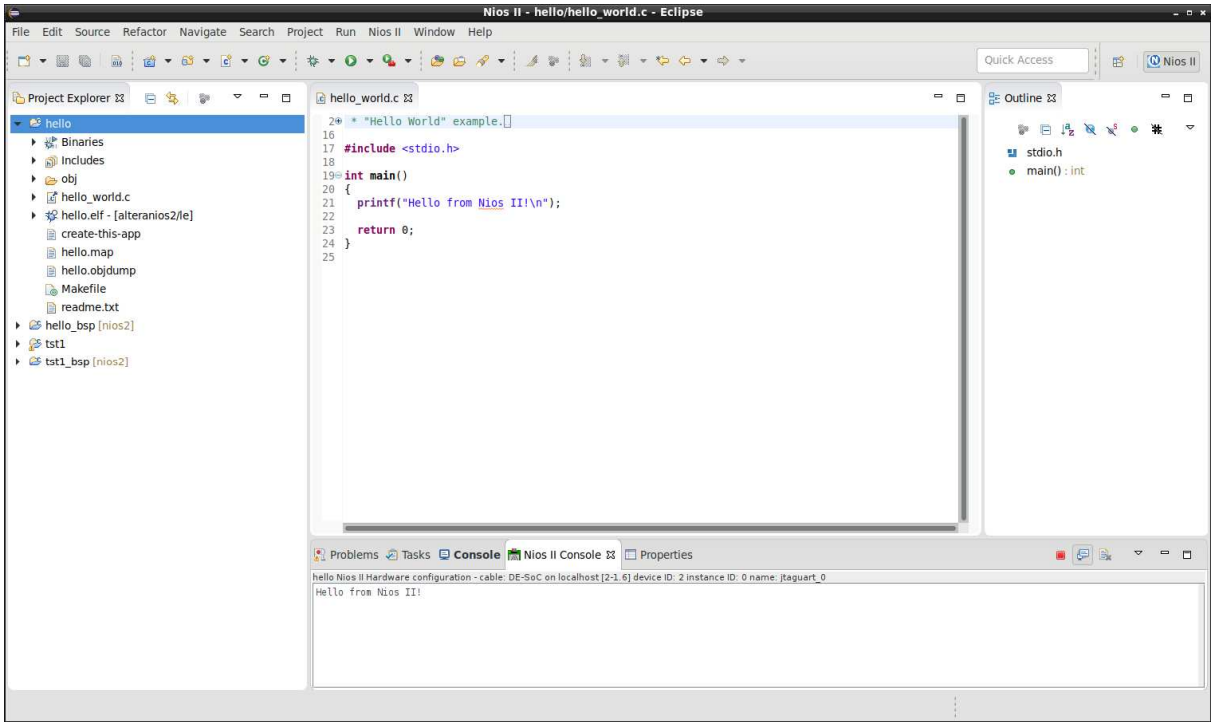
- On a donc 2 projets : *hello* et *hello_bsp*. Dans le projet *hello*, on ouvrira le fichier `hello_world.c` qui est le programme principal qui sera exécuté par le processeur NIOS II du système SoPC.

- Analyser le source du fichier `hello_world.c`.

- Compiler le projet *hello* par le menu *Project > Build Project*. On peut le faire aussi par le menu contextuel avec le clic droit de la souris sur le projet *hello* dans la fenêtre *Project Explorer*.

- Télécharger le code exécutable (fichier `.elf`) par le menu contextuel du projet *hello* (clic droit de la souris sur le projet *hello*) en choisissant le menu *Run As > 3 Nios II Hardware* ou en cliquant sur le bouton vert avec le triangle blanc  .

- Remarquer les traces d'exécution dans la fenêtre *Nios II Console* qui correspond à ce que le PC de développement reçoit sur la liaison JTAG/UART :



Traces d'exécution du projet *Eclipse Hello*

- Par le projet *hello*, quels blocs IP du système SoPC a-t-on fonctionnellement validé ?

5. EX 3 INTEL : CREATION DU BSP

Quand nous avons créé le projet *hello*, *Eclipse* nous a créé automatiquement le projet *hello_bsp*.

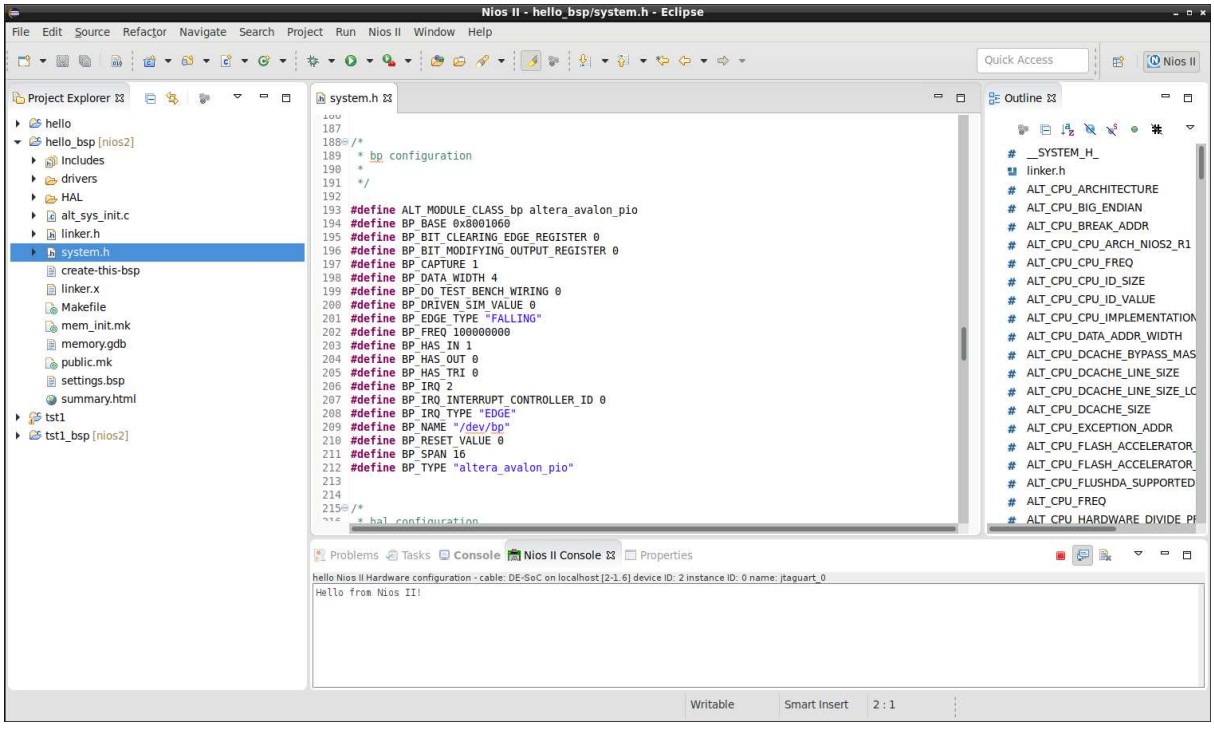
Le projet *hello_bsp* est un BSP support qui correspond à la couche d'abstraction matérielle de base appelée HAL (*Hardware Abstraction Layer*).

Cela correspond à un ensemble de fonctions C générées de façon automatique qui permet d'interagir avec les blocs IP du système SoPC. On retrouve dans ces routines `printf()` par la liaison JTAG/UART par exemple.

On peut avoir un aperçu de la HAL en regardant dans le répertoire *HAL > src* du projet *hello_bsp*.

Un fichier important du projet *hello_bsp* est le fichier C `system.h` qui est la traduction en langage C du fichier `.sopcinfo`. On le voit sur la figure suivante, c'est une suite de `#define` qui donne les caractéristiques essentielles pour chaque bloc IP.

Sur la figure, on voit par exemple les caractéristiques du bloc IP *bp*. On voit que l'adresse de base des registres du bloc IP *bp* correspond à l'étiquette `BP_BASE`.



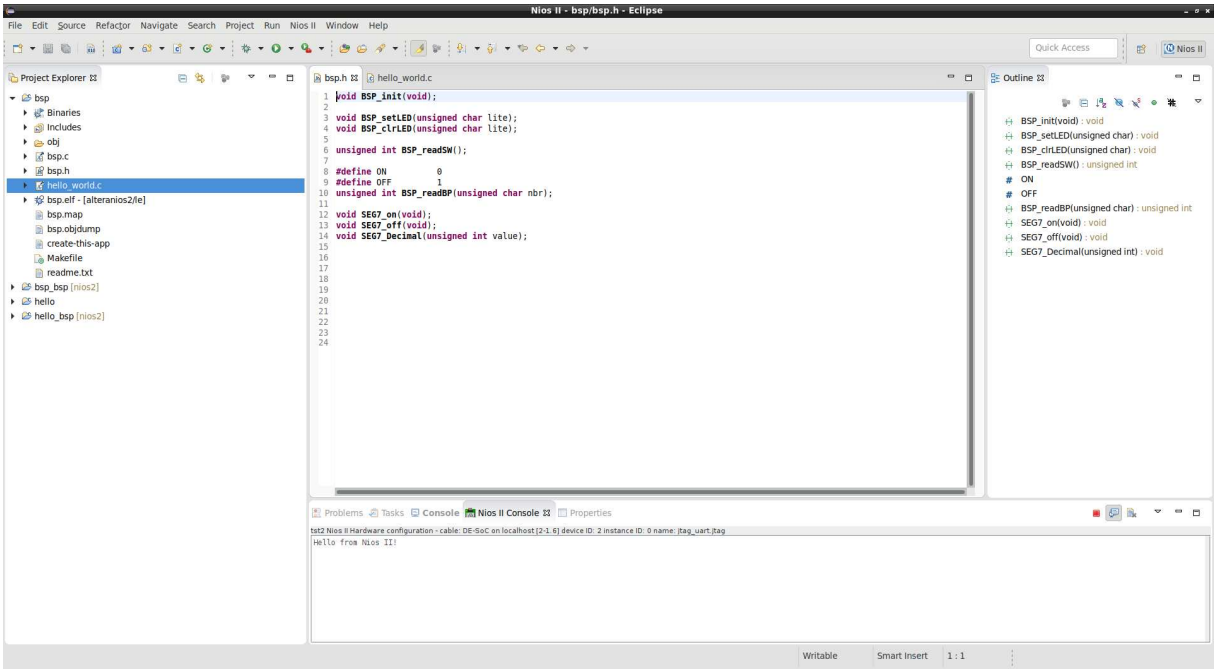
Contenu du fichier `system.h` du projet *Eclipse Hello*

L'idée est de construire notre propre BSP au dessus de la HAL pour le pilotage des blocs IP *bp*, *switchs*, *leds* et *seg7*.

On utilisera pour cela les routines d'E/S de la HAL.

- Créer un projet nommé *bsp* à partir du *template Hello Word*.

- Importer les fichiers ressources `bsp.h`, `bsp.c` et `hello_world.c` du répertoire `de10/ressources/` dans le projet *bsp*. On peut faire cela avec l'explorateur de fichiers sous Linux et faire un copier/coller des 3 fichiers dans le projet *bsp* par un *drag and drop*. On obtient la figure suivante :



Projet *Eclipse bsp*

Pour réaliser notre BSP, on va utiliser 2 fonctions de base fournies par la HAL pour la lecture et l'écriture des registres de nos périphériques.

On a à disposition dans le fichier `io.h` les fonctions :

```
IORD(BASE, REGNUM);
IOWR(BASE, REGNUM, DATA);
```

`IORD()` renvoie le contenu 32 bits du registre numéro `REGNUM` à partir de l'adresse de base `BASE`.
`IOWR()` écrit la donnée 32 bits `DATA` dans le registre numéro `REGNUM` à partir de l'adresse de base `BASE`.

Notons aussi, que l'on relit exactement ce que l'on a écrit précédemment…

Les adresses de base de nos périphériques sont définies dans le fichier `system.h` de la HAL et sont pour nous :

- `LEDS_BASE` pour les leds.
- `SWITCHS_BASE` pour les switchs.
- `BP_BASE` pour les boutons poussoir.
- `SEG7_BASE` pour les afficheurs 7 segments.

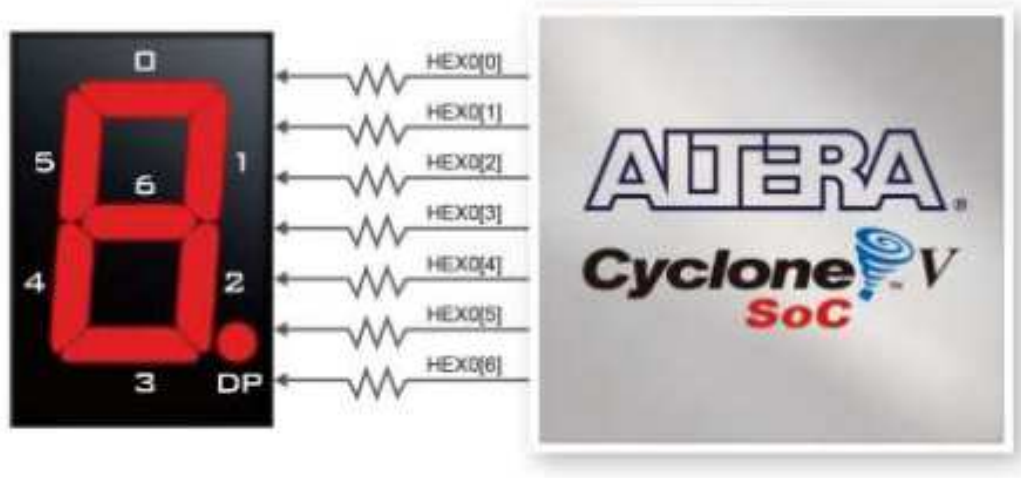
Le registre 0 correspond à la donnée à lire ou à écrire pour les leds, switchs ou boutons poussoir.

Le bit 0 du registre 0 correspond à la led 0, le switch 0 ou le bouton poussoir 0

Ces 3 périphériques sont basés sur le même bloc IP et fonctionnent donc de la même façon.

Pour les afficheurs 7 segments, le registre 0 correspond au masque à écrire pour l'afficheur 0 jusqu'au registre 6 qui correspond au masque à écrire pour l'afficheur 6.

Les bits du registre d'un afficheur 7 segments sont connectés comme indiqué sur la figure suivante. Le point décimal n'est pas câblé :



Câblage des afficheurs 7 segments

Ce périphérique est basé sur un bloc IP différent de celui des leds par exemple.

Dans le fichier `bsp.c`, on a défini la fonction :

```
SEG7_SET(seg7_nbr, seg7_mask);
```

Cette fonction permet d'écrire simplement la valeur du masque `seg7_mask` dans le registre 0 de l'afficheur numéro `seg7_nbr`. Il y a `SEG7_NUM` afficheurs soit ici 6.

Ces explications sont alors suffisantes pour écrire notre BSP.

Les fonctions du BSP à développer sont :

- `void BSP_init(void)` : initialisation de la carte. Extinction des leds et des afficheurs 7 segments.
- `void BSP_setLED(unsigned char lite)` : allumage de la led numéro `lite`.
- `void BSP_clrLED(unsigned char lite)` : extinction de la led numéro `lite`.
- `unsigned int BSP_readSW()` : lecture de la valeur courante des 10 switchs.
- `unsigned int BSP_readBP(unsigned char nbr)` : valeur courante (on ou off) du bouton poussoir numéro `nbr` (de 0 à 3, 3 étant le bouton poussoir le plus à gauche).
- `void SEG7_on(void)` : allumage complet des segments des 6 afficheurs 7 segments.
- `void SEG7_off(void)` : extinction des 6 afficheurs 7 segments.
- `void SEG7_Decimal(unsigned int value)` : affichage sur les afficheurs 7 segments de la valeur décimale `value`.

- Modifier le fichier `bsp.c` pour implémenter le BSP.

- Modifier le fichier principal `hello_world.c` pour tester séquentiellement (pas de multitâche pour l'instant) chaque périphérique.

- Le système SoPC est-il finalement fonctionnel ?

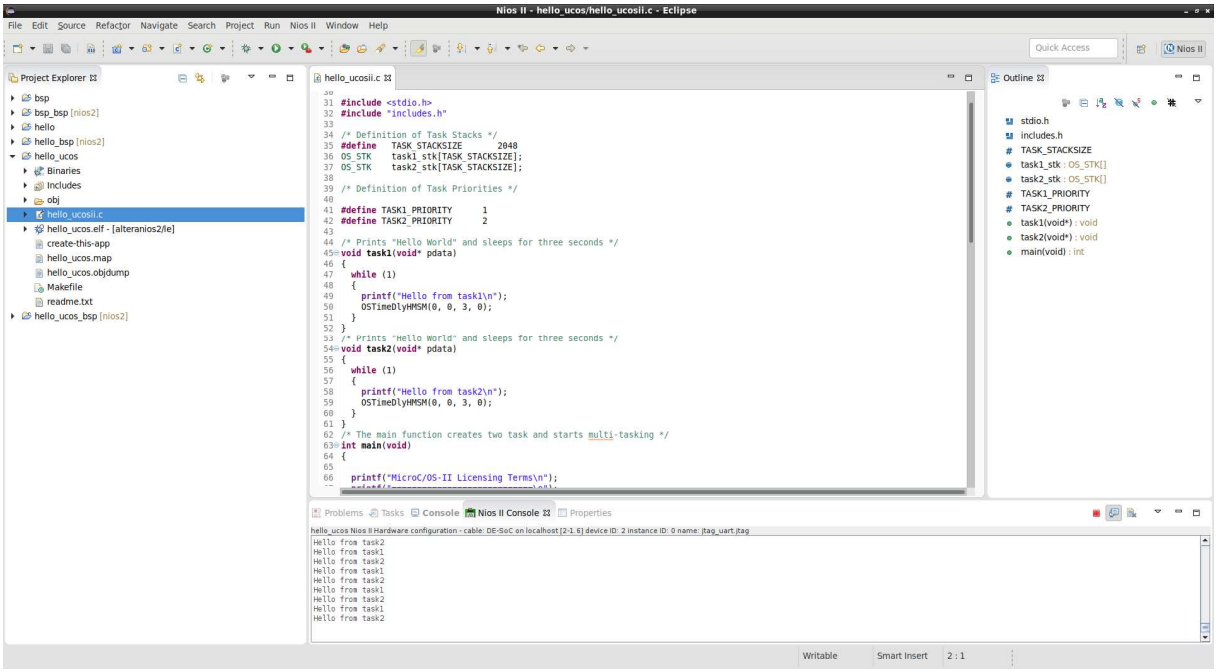
6. EX 4 INTEL : HELLO WORLD µC/OS II

On va s'intéresser maintenant au noyau Temps Réel µC/OS II qui est supporté par le processeur NIOS II et accessible avec *Eclipse*.

Outre le support du Temps Réel dur, µC/OS II nous apporte aussi le multitâche.

On va d'abord créer le projet µC/OS II « *Hello World* » qui consiste à la création de 2 tâches périodiques qui écrivent sur la liaison JTAG/UART.

- Créer un nouveau projet *Eclipse hello_ucos* en utilisant le *template Hello MicroC/OS-II*.
- Tester. On obtient la figure suivante :



Projet *hello_ucos*

- Quel périphérique (bloc IP) est essentiel pour faire fonctionner µC/OS II ?

7. EX 5 INTEL : TESTS DES PERIPHERIQUES SOUS µC/OS II

On désire créer un projet *Eclipse* qui teste en parallèle l'ensemble des périphériques sous µC/OS II.

- Créer le projet Eclipse *tst_all* à partir du *template Hello MicroC/OS-II*. On importera bien sûr les fichiers `bsp.h` et `bsp.c` de notre BSP dans le projet.

- Créer :
  - Une tâche qui réalise un chenillard sur les 10 leds.
  - Une tâche qui renvoie l'état courant des 10 switchs.
  - Une tâche qui détecte l'appui sur un bouton poussoir.
  - Une tâche qui affiche sur les 6 afficheurs 7 segments une valeur décimale qui s'incrémente chaque seconde.

- Tester.

8. EX 6 INTEL : MINIPROJET : CHRONOMETRE ET HORLOGE

On désire créer un chronomètre au dixième de seconde qui affiche le temps sur les 6 afficheurs 7 segments.

L'afficheur 1 affichera les dixièmes de seconde (l'afficheur 0 le plus à droite reste à 0) alors que les afficheurs 1 à 5 affichent les secondes.

Le bouton poussoir 0 est le start/stop du chronomètre.

Le bouton poussoir 3 est le reset du chronomètre.

- Créer le projet *Eclipse miniprojet* à partir du *template Hello MicroC/OS-II*.
- Développer le miniprojet.
- Tester. Remarquer l'existence de rebonds sur les boutons poussoir si l'on ne fait rien pour les supprimer.

Supplément :

- Réaliser une horloge qui affiche heures, minutes, secondes sur les 6 afficheurs 7 segments. Le bouton poussoir *bp3* réglera les dizaines d'heures, le bouton poussoir *bp2* les heures, le bouton poussoir *bp1* les dizaines de minutes et le bouton poussoir *bp0* les minutes.

9. EX 7 INTEL : INTEGRATION D'UN PERIPHERIQUE MATERIEL LIBRE

9.1. Introduction

L'objectif de ce TP est d'intégrer un périphérique matériel Libre, en l'occurrence une interface VGA qui inclut le contrôleur VGA et sa mémoire pixels mais aussi un générateur de caractères et sa mémoire caractères.

Cette interface VGA est issue du programme universitaire Intel et est libre d'usage.

On analysera dans un premier temps la structure du système SoPC avec l'outil *Platform Designer* puis on réalisera la synthèse avec l'outil *Quartus Prime*.
Dans un second temps, on testera le contrôleur VGA et sa mémoire pixels. On utilisera pour cela une bibliothèque maison `vgalib` qui permet de piloter la mémoire pixels, de dessiner des rectangles mais aussi des caractères en mode pixels. Pour cela, on utilisera une police de caractères ou fonte pour implanter la page de codes 437 d'IBM. A partir de cela, différents points seront abordés via les exercices proposés.

9.2. Intégration d'un périphérique. Interface VGA

L'interface VGA (*Video Graphics Adapter*) est une norme de signal vidéo que l'on trouve principalement dans les ordinateurs individuels. Ce signal vidéo est constitué de 5 signaux, 3 analogiques avec des niveaux de 0.7 à 1.0 V pour chacune des couleurs Rouge, Vert et Bleu (RGB) accompagnés de 2 signaux logiques que sont les synchros horizontale et verticale.

Pour le VGA standard, l'écran contient 640 par 480 pixels qui doivent être balayés à une fréquence supérieure à 30 Hz si l'on veut éviter un effet de scintillement. On trouve en général sur les PC une fréquence de rafraîchissement de 60 Hz (60 trames par seconde).
Les écrans du commerce fonctionnent tous dans une certaine plage autour de cette fréquence de référence. Si l'on veut évaluer approximativement la fréquence pixel, il suffit de diviser 1/60 par le produit 680x480. Cela donne 50 ns environ entre 2 pixels.

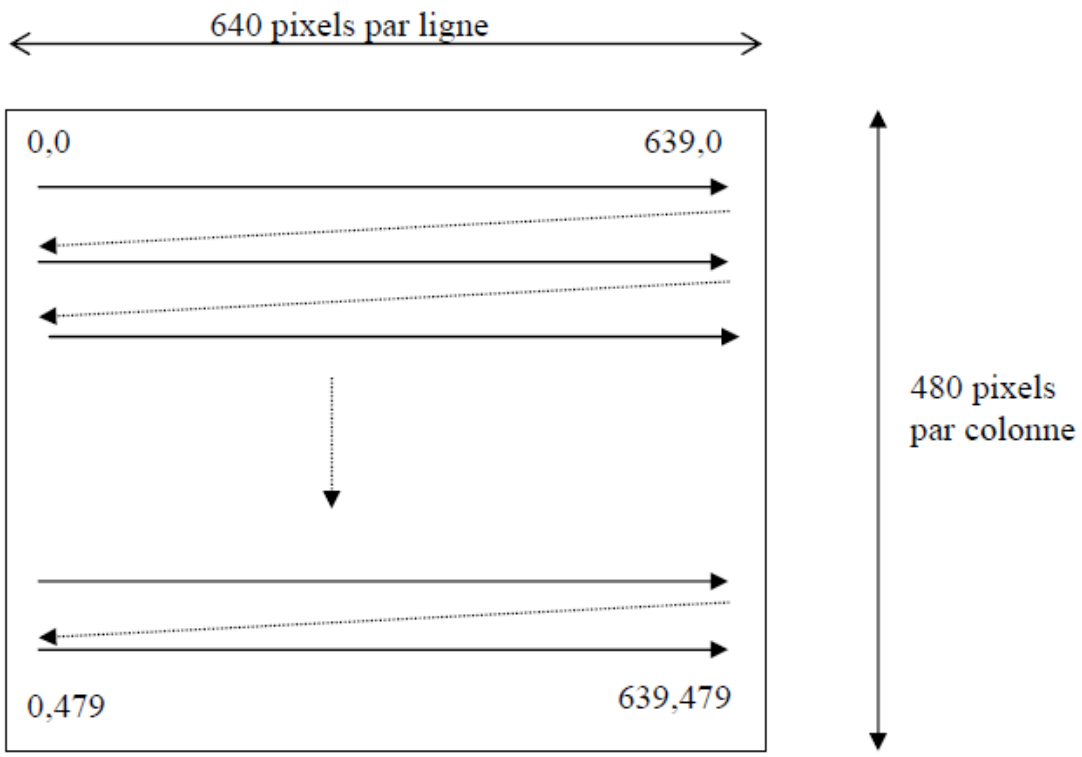
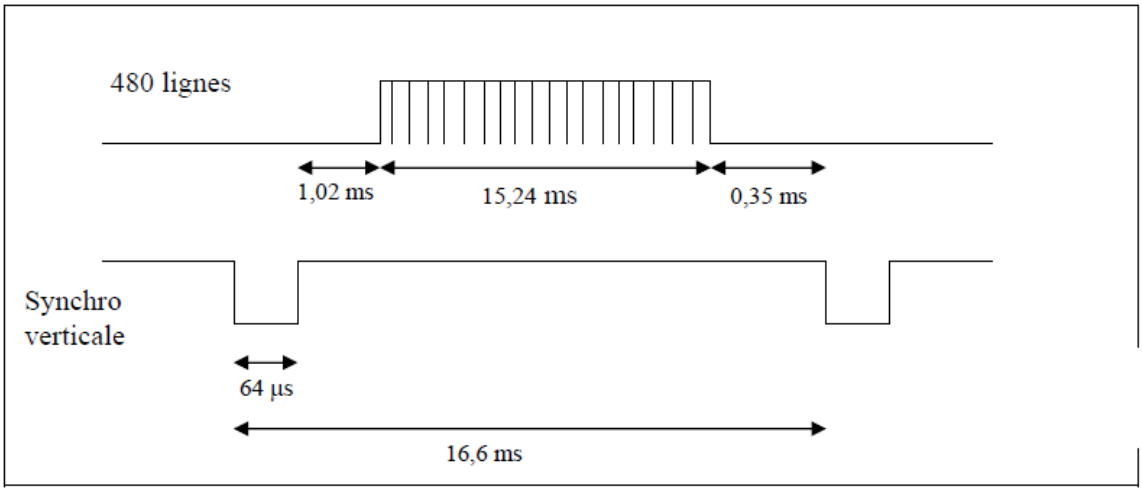


Image VGA 640x480

La trame VGA est découpée en 525 lignes.

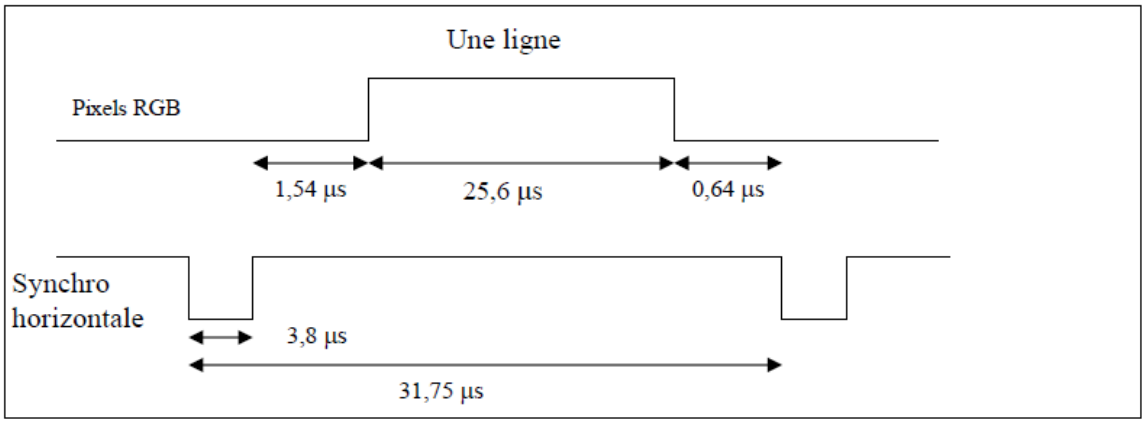
Une fréquence image de 60 Hz entraîne alors une fréquence ligne de 31,5 kHz. Une ligne correspond pour une fréquence pixel à 25 MHz à un comptage de 794 points comprenant les 640 points (pixels) utiles.

Le balayage vertical se décompose en une synchro (2 lignes) suivie de 32 lignes noires suivies des 480 lignes de pixels utiles suivies enfin de 11 lignes noires (soit 2+32+480+11=525 lignes au total).



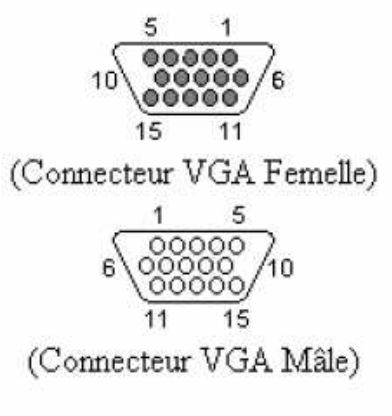
Timing de la synchronisation verticale à 60 Hz

Le balayage horizontal se décompose en un signal de synchronisation (95 points), un pallier avant noir (43 points), les 640 pixels et un pallier arrière noir (16 points) (soit 95+43+640+16=794 points au total).



Timing de la synchronisation horizontale à 31,5 kHz

Le connecteur VGA permet de relier le système générateur de vidéo (le plus souvent l'ordinateur) à l'écran.



Connecteur VGA

Le brochage du connecteur VGA est le suivant :

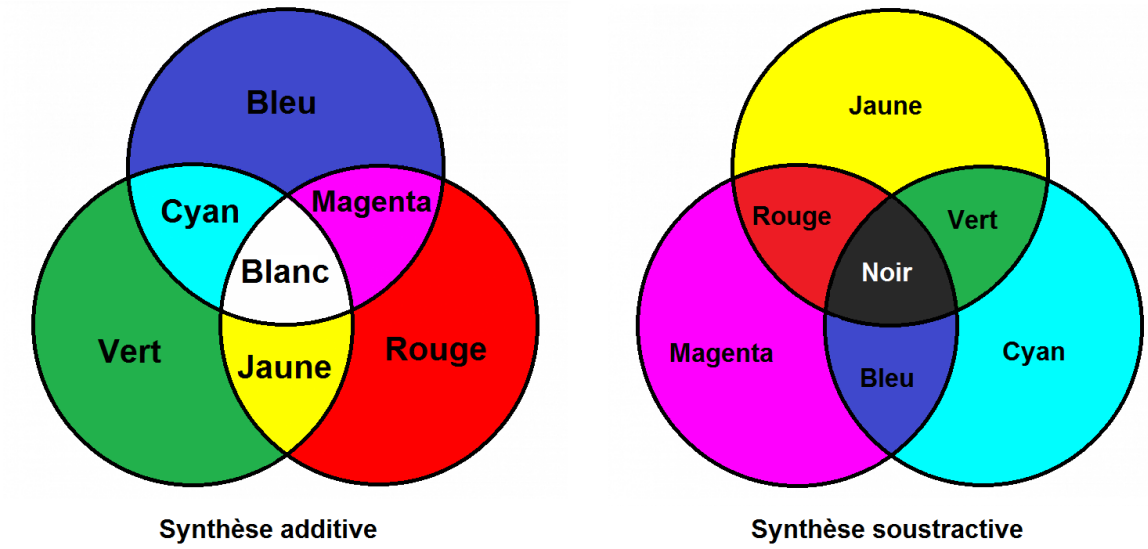
Broche	Nom	Description
1	RED	Vidéo rouge (75 ohms, 0.7 volt)
2	GREEN	Vidéo verte (75 ohms, 0.7 volt)
3	BLUE	Vidéo bleu (75 ohms, 0.7 volt)
4	RES	Réservé
5	GND	Masse
6	RGND	Masse rouge
7	GGND	Masse verte
8	BGND	Masse bleu
9	+5V	+5V continu
10	SGND	Masse synchro
11	ID0	Monitor ID Bit 0 (optionnel)
12	SDA	Ligne de données série DDC
13	HSYNC ou CSYNC	Synchro horizontale ou composite
14	VSYNC	Synchro verticale
15	SCL	Ligne d'horloge DDC (Display Data Channel)

Signaux du connecteur VGA

Il convient aussi d'aborder quelques notions sur les couleurs.

Il existe d'abord pour la synthèse de couleurs la synthèse additive et la synthèse soustractive.
La synthèse additive est utilisée en informatique mais aussi en peinture ou en photographie.
La synthèse soustractive est utilisée en imprimerie mais aussi en peinture.

La figure suivante présente les 2 synthèses :



Synthèse additive et synthèse soustractive des couleurs

(d'après https://www.methodephysique.fr/couleurs_synthese_additive_soustractive/)

On distingue alors différents types de couleurs ;

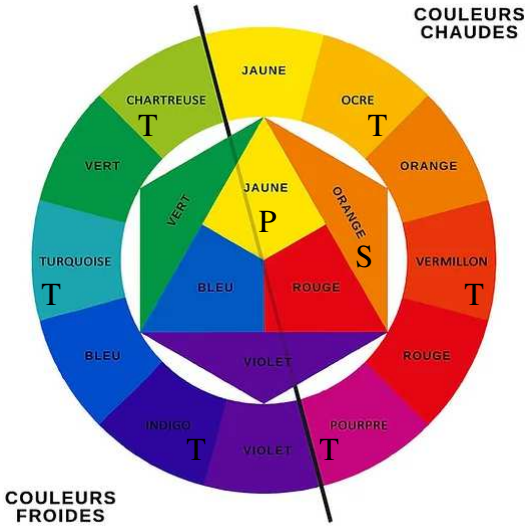
- Une couleur primaire est une couleur qui ne peut pas être créée par le mélange d'autres couleurs. En synthèse additive (synthèse RVB), les trois couleurs primaires sont le rouge, le vert et le bleu. Le mélange de ces 3 couleurs donne le blanc. En synthèse soustractive (synthèse CMJ), les trois couleurs primaires sont le magenta (rouge rosé), le cyan (bleu clair) et le jaune. Le mélange de ces 3 couleurs donne le noir.

- Une couleur secondaire est obtenue en mélangeant à part égale deux couleurs primaires entre elles. En synthèse additive, on obtient la couleur secondaire jaune en mélangeant le rouge et le vert, la couleur secondaire magenta en mélangeant le rouge et le bleu et la couleur secondaire cyan en mélangeant le bleu et le vert. En synthèse soustractive, on obtient la couleur secondaire bleu en mélangeant le magenta et le cyan, la couleur secondaire verte en mélangeant le cyan et le jaune et la couleur secondaire rouge en mélangeant le jaune et le magenta.

- Une couleur complémentaire est une couleur qui, ajoutée à une autre couleur primaire, donnera du blanc en synthèse additive ou du noir en synthèse soustractive. Les couleurs complémentaires sont les couleurs qui se trouvent dans une position opposable dans le cercle chromatique. Ainsi, la couleur complémentaire d'une couleur primaire est la couleur qui résulte du mélange des deux autres couleurs primaires. La couleur complémentaire d'une couleur secondaire est la couleur primaire qu'elle ne contient pas. En synthèse additive, la couleur complémentaire de la couleur primaire verte est le magenta, la couleur complémentaire de la couleur primaire rouge est le cyan et la couleur complémentaire de la couleur primaire bleu est le jaune. En synthèse soustractive, la couleur complémentaire de la couleur primaire magenta est le vert, la couleur complémentaire de la couleur primaire cyan est le rouge et la couleur complémentaire de la couleur primaire jaune est le bleu. C'est exactement le même résultat qu'avec la synthèse additive.

En peinture ou en photographie, on utilise un cercle chromatique des couleurs. Le cercle chromatique des couleurs a été inventé par Johannes Itten (1888-1967) et rassemble 12 couleurs (couleurs **P**rimaires, couleurs **S**econdaires et couleurs **T**ertiaires). Les couleurs complémentaires sont très utilisées dans ces arts car elles se subliment réciproquement. Elles se trouvent opposées sur le cercle chromatique.

Les couleurs complémentaires sont réévaluées pour être plus faciles à générer. Les couleurs complémentaires (cyan, rouge), (magenta, vert,) et (jaune, bleu) sont complétées/remplacées sur le cercle chromatique par les couleurs (bleu, orange), (rouge, vert) et (jaune, violet).



Cercle chromatique en synthèse soustractive
(d'après https://www.peinture78.com/post/comprendre-les-couleurs-et-le-cercle-chromatique)

Une photographie mélangeant le couple (orange, bleu) aura un fort impact visuel comme le montre la photographie suivante :

Couleurs complémentaires (orange, bleu) du martin pêcheur et impact visuel

(crédit photo : patrice kadionik)

En informatique, on utilise la synthèse additive RVB. L'interface VGA que l'on utilise respecte cette représentation des couleurs RVB (ou RGB en anglais). L'encodage RGB classique est du type 888 soit 8 bits pour le rouge, 8 bits pour le vert et 8 bits pour le bleu soit 24 bits en tout.

Notre interface VGA utilise l'encodage RGB 565 soit 5 bits pour le rouge, 6 bits pour le vert et 5 bits pour le bleu soit 16 bits en tout. Une couleur est donc codée sur 16 bits au lieu de 24 bits.

Pour afficher un pixel noir, la valeur à utiliser est 0x0000. Pour afficher un pixel blanc, la valeur à utiliser est 0xffff.

En imprimerie, on utilise la synthèse soustractive CMJ. Pour générer le noir, il faudrait utiliser en même temps le cyan, le magenta et le jaune en quantité égale pour obtenir un noir plus ou moins gris. On ajoute alors une 4ème cartouche de noir pour imprimer le noir et économiser les 3 autres cartouches. On a donc ici une synthèse soustractive CMJN.

9.3. Synthèse du système SoPC

- Se placer dans le répertoire `video_nios/`. Nous allons maintenant construire notre *design* de référence *DE10_Standard_golden_top* :

```
host% cd de10/video_nios
host% ls
Char_Buf_Subsystem.qsys        DE10_Standard_golden_top.qsf  ip/
DE10_Standard_golden_top.htm   DE10_Standard_golden_top.sdc  nios2.qsys
DE10_Standard_golden_top.qpf   DE10_Standard_golden_top.v
VGA_Subsystem.qsys
```

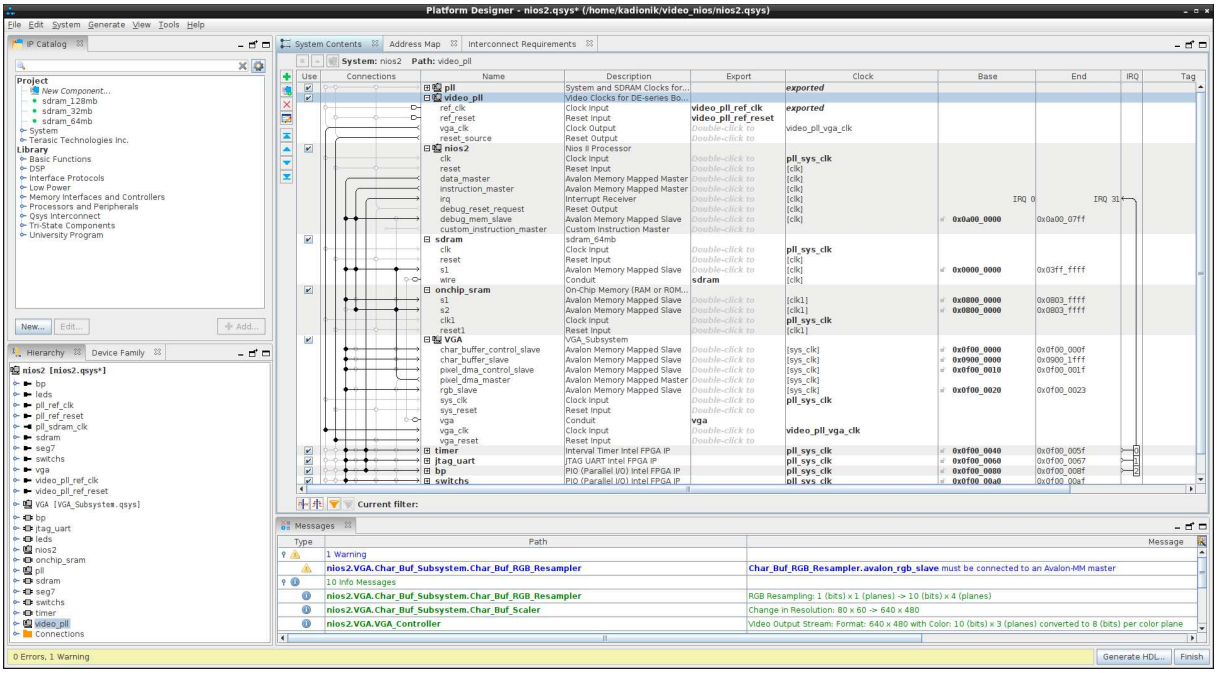
On remarque que l'on a pour ce *design* 3 fichiers `.qsys` pour *Platform Designer*. Le fichier principal est `nios2.qsys`, les 2 fichiers `VGA_Subsystem.qsys` et `Char_Buf_Subsystem.qsys` étant inclus dans le fichier `nios2.qsys`. Le fichier `VGA_Subsystem.qsys` correspond à l'interface VGA tandis que le fichier `Char_Buf_Subsystem.qsys` correspond au sous-système d'affichage de caractères inclus dans l'interface VGA. On pourra donc travailler au niveau pixel avec une mémoire vidéo pixels ou bien afficher des caractères avec une mémoire vidéo caractères. On n'utilisera par la suite que la mémoire vidéo pixels.

- Se placer dans l'environnement de développement Intel puis lancer *Quartus Prime* :

```
host% n2sdk
[Xilinx EDK]$ quartus
```

- Ouvrir le projet *Quartus Prime DE10_Standard_golden_top* par le menu *File > Open Project...* (ouverture du fichier `DE10_Standard_golden_top.qpf`).

- Lancer ensuite *Platform Designer* par le menu *Tool > Platform designer*. Ouvrir le fichier `nios2.qsys`. On obtient la figure suivante :



Fichier `nios2.qsys` pour le design *video_nios*

- Quelle sont les adresses de début et de fin de la mémoire SDRAM ?
- Quelle sont les adresses de début et de fin de la mémoire vidéo pixels ?
- Quelle sont les adresses de début et de fin de la mémoire vidéo caractères ?
- Est-ce que le *design standard_nios* est inclus dans le *design video_nios* ?
- Réaliser la génération HDL du système SoPC *DE10_Standard_golden_top* (bouton *Generate*) puis cliquer sur le bouton *Finish* pour sortir de *Platform Designer*.
- Analyser le fichier `DE10_Standard_golden_top.v` qui devient alors :

```
module DE10_Standard_golden_top(

      ///////// CLOCK /////////
      input              CLOCK2_50,
      input              CLOCK3_50,
      input              CLOCK4_50,
      input              CLOCK_50,

. . .

wire  HEX0P;
wire  HEX1P;
wire  HEX2P;
wire  HEX3P;
wire  HEX4P;
wire  HEX5P;
```

```
nios2 u0 (
.pll_ref_clk_clk          (CLOCK_50),
.pll_ref_reset_reset      (1'b0),

.pll_sdram_clk_clk        (DRAM_CLK),
.sdram_addr               (DRAM_ADDR),
.sdram_ba                 (DRAM_BA),
.sdram_cas_n              (DRAM_CAS_N),
.sdram_cke                (DRAM_CKE),
.sdram_cs_n               (DRAM_CS_N),
.sdram_dq                 (DRAM_DQ),
.sdram_dqm                ({DRAM_UDQM,DRAM_LDQM}),
.sdram_ras_n              (DRAM_RAS_N),
.sdram_we_n               (DRAM_WE_N),

.bp_export                (KEY),
.switchs_export           (SW),
.leds_export              (LEDR),
.seg7_export   ({HEX5P,HEX5,HEX4P,HEX4,HEX3P,HEX3,HEX2P,HEX2,HEX1P,HEX1,HEX0P,HEX0}),

// VGA Subsystem
.video_pll_ref_clk_clk                      (CLOCK2_50),
.video_pll_ref_reset_reset             (1'b0),

.vga_CLK                              (VGA_CLK),
.vga_BLANK                            (VGA_BLANK_N),
.vga_SYNC                             (VGA_SYNC_N),
.vga_HS                               (VGA_HS),
.vga_VS                               (VGA_VS),
.vga_R                                (VGA_R),
.vga_G                                (VGA_G),
.vga_B                                (VGA_B)
  );

endmodule
```

- Qu'a-t-on rajouté en plus dans le fichier `DE10_Standard_golden_top.v` ?

- Réaliser la synthèse du système SoPC. Après synthèse, quelles sont les ressources consommées et combien dans le circuit FPGA ?

- Programmer le circuit Cyclone V de la carte DE-Standard.

Nous en avons alors fini avec la partie matérielle de construction de notre système SoPC.

Il nous reste la deuxième partie du *codesign* qui correspond à la programmation logicielle.

On pourra toujours reprogrammer le circuit FPGA depuis *Eclipse*.

On peut enfin fermer *Quartus Prime*...

10. EX 8 INTEL : HELLO WORLD µC/OS II

Nous allons réaliser un premier test logiciel fonctionnel en implémentant sur la carte cible DE10-Standard le programme « *Hello World* ».

On utilisera par la suite *Eclipse* comme atelier de développement logiciel IDE.

On utilisera pour cela le *shell script* `goeclipe` qui lancera *Eclipse* avec les bons paramètres.

Eclipse utilise un espace de travail qui est généralement le répertoire `software/` que l'on a sous `de10/video_nios/`.

- Se placer ensuite dans le répertoire `de10/video_nios` :
  ```
  host% cd de10/standard_nios
  ```

- Se placer dans l'environnement de développement Intel si ce n'est pas déjà fait puis lancer *Eclipse*. On ajustera le *Workspace* à `de10/video_nios/software/` :
  ```
  host% n2sdk
  [Xilinx EDK]$ goeclipse
  ```

- Créer un nouveau projet *Eclipse hello_ucos* en utilisant le *template Hello MicroC/OS-II*.

- Importer dans le projet *hello_ucos* les fichiers `bsp.h`, `bsp.c` créés pour le *design standard_nios*.

- Importer dans le projet *hello_ucos* les fichiers `hello_ucosii.c`, `rom8x8.c`, `vgalib.c` et `vgalib.h` du répertoire `de10/ressources/` dans le projet *hello_ucos*.

- Tester.

Par la suite, dans tous les projets *Eclipse* suivants, on incorporera dans le projet les fichiers `bsp.h`, `bsp.c`, `hello_ucosii.c` (que l'on modifiera suivant l'exercice), `rom8x8.c`, `vgalib.c` et `vgalib.h` du projet *hello_ucos*…

11. EX 9 INTEL : AFFICHAGE DE RECTANGLES

On désire créer un projet *Eclipse* qui affiche des rectangles de taille et de couleur aléatoire. Un nouveau rectangle est affiché à chaque seconde.

On pourra utiliser la fonction pseudo-aléatoire `rand()`.
On utilisera la bibliothèque VGA contenue dans le fichier `vgalib.c`.

La bibliothèque VGA utilise la mémoire vidéo pixels mais ne met pas en œuvre la mémoire vidéo caractères. Les caractères à afficher seront directement écrits dans la mémoire vidéo pixels sous forme d'une matrice de 8 par 8 pixels et sa représentation mémorisée dans une table de 8 octets définie (tableau `rom8x8_bits`) dans le fichier `rom8x8.c` .

Chaque bit d'un octet représente un pixel. Par exemple, pour caractère A, nous avons :

```
00111000
01101100
11000110
11000110
11111110
11000110
11000110
00000000
```

Soit dans le fichier `rom8x8.c` :

```
. . .
/* Character A (0x41):
   ht=8, width=8
   +--------+
   |   ***  |
   |  ** **  |
   | **   ** |
   | **   ** |
   | ******* |
   | **   ** |
   | **   ** |
   |        |
   +--------+ */
0x3800,
0x6c00,
0xc600,
0xc600,
0xfe00,
0xc600,
0xc600,
0x0000,
. . .
```

L'interface VGA a une résolution de 640x480 pixels. Pour réduire la consommation mémoire dans le circuit FPGA, la résolution est divisée par 2 en abscisse (x) et par 2 en ordonnée (y), ce qui donne une résolution effective de 320x240 pixels. Pour respecter le standard VGA, un pixel de la mémoire vidéo pixels est dupliqué sur l'interface VGA en x et en y, ce qui donnera alors un carré sur l'écran de 2x2 pixels.
Pour les caractères 8x8 pixels à afficher, ils seront affichés à l'écran sous forme de caractères 16x16 pixels. Il y a donc un agrandissement d'un facteur 2.
On pourra ainsi afficher à l'écran 30 lignes de 40 caractères par ligne. La bibliothèque gère la position courante pixel et en partie la position courante caractère.

Les fonctions de la bibliothèque VGA sont :

- `void VgaInit(void)` : initialisation de la bibliothèque VGA notamment la position courante pixel.
- `void VgaClearScreen(void)` : effacement de la mémoire vidéo pixels et de la mémoire caractères.
- `void VgaSetCursor(int col, int row)` : spécification de la position caractère.
- `void VgaWriteCharacter(unsigned char character)` : affichage/écriture d'un caractère à la position courante caractère.
- `void VgaWriteScreen(char * string)` : affichage à partir de la position courante caractère d'une chaîne de caractères. La position caractère est incrémentée automatiquement mais le retour à la ligne suivante n'est pas implémenté.
- `void VgaPrintAt(int col, int row, char * string)` : affichage d'une chaîne de caractères à la position caractère spécifiée mais le retour à la ligne suivante n'est pas implémenté.
- `int VgaDrawBox(int x1, int y1, int x2, int y2, short pixel_color)` : affichage d'un rectangle de couleur 16 bits au format 565 dont le coin supérieur gauche est (x1, y1) et le coin inférieur bas est (x2, y2).

- Créer un nouveau projet *Eclipse rectangle* en utilisant le *template Hello MicroC/OS-II*.

- Importer dans le projet *rectangle* les fichiers `bsp.h`, `bsp.c`, `hello_ucosii.c`, `rom8x8.c`, `vgalib.c` et `vgalib.h` du projet *hello_ucos*.

- Tester.

On obtient alors la figure suivante :

Exercice *rectangle*

12. EX 10 INTEL : AFFICHAGE DE LA POLICE DE CARACTERES

Le fichier `rom8x8.c` implémente une police de caractères ou fonte pour implanter la page de codes 437 d'IBM.

Une page de code est un standard informatique qui vise à donner un numéro à chaque caractère d'une langue. Elle constitue donc une méthode simple de codage des caractères (https://fr.m.wikipedia.org/wiki/Page_de_code).

La page de code 437 est une page de code définie par IBM et qui était utilisée aux États-Unis avec le système DOS et d'autres systèmes de la même époque et a été la première définie matériellement sur les cartes d'affichage VGA des premiers PC fabriqués par IBM (https://fr.m.wikipedia.org/wiki/Page_de_code_437).

Nous allons afficher à l'écran cette police de caractères.

- Créer un nouveau projet *Eclipse fonte* en utilisant le *template Hello MicroC/OS-II*.
- Tester.

On obtient alors la figure suivante :

Exercice *fonte*

13. EX 11 INTEL : MINIPROJET : CHRONOMETRE ET HORLOGE

On reprend l'exercice 6. On désire créer un chronomètre au dixième de seconde qui affiche le temps sur les 6 afficheurs 7 segments et sur l'écran.

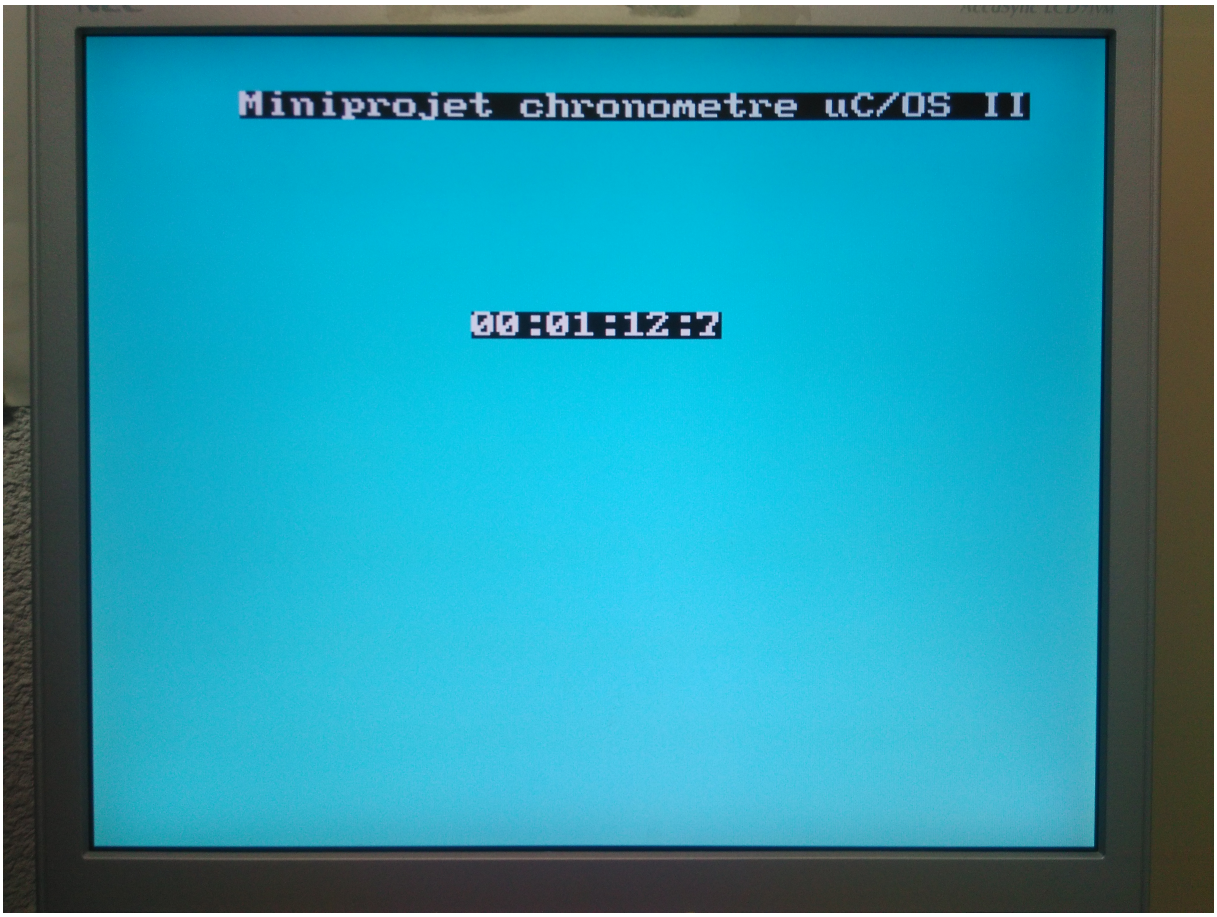
L'afficheur 1 affichera les dixièmes de seconde (l'afficheur 0 le plus à droite reste à 0) alors que les afficheurs 1 à 5 affichent les secondes.

Le bouton poussoir 0 est le start/stop du chronomètre.

Le bouton poussoir 3 est le reset du chronomètre.

- Créer le projet *Eclipse chrono* à partir du *template Hello MicroC/OS-II*.
- Tester.

On obtient alors la figure suivante :



Miniprojet *chrono*

Supplément :

- Réaliser une horloge qui affiche heures, minutes, secondes sur les 6 afficheurs 7 segments et sur l'écran. Le bouton poussoir *bp3* réglera les dizaines d'heures, le bouton poussoir *bp2* les heures, le bouton poussoir *bp1* les dizaines de minutes et le bouton poussoir *bp0* les minutes.

- On pourra créer un nouveau miniprojet *horloge* pour cela. On pourra aussi rajouter l'affichage de rectangles de taille et de couleur aléatoires qui apparaissent à l'écran à un moment aléatoire qui est décorrélé de l'affichage de l'heure.

On obtient alors la figure suivante :

Miniprojet *horloge*

14. GRAND TP 2 : MISE EN ŒUVRE DU SOPC AVEC AMD ZYNQ

14.1. Introduction

Il s'agit d'utiliser l'environnement de développement SoPC d'AMD qu'il s'agisse de Vivado comme outil de synthèse VHDL et de placement routage ou de l'environnement Linux comme environnement de développement croisé pour la partie logicielle.

Ce deuxième « grand » TP se décompose en :

- L'intégration d'un périphérique (fonction compteur d'un *timer* 64 bits simplifié) dans un SoPC pour circuit FPGA Zynq avec l'outil AMD Vivado.
- La mise en œuvre de Linux embarqué sur le processeur ARM Cortex-A9 du circuit FPGA Zynq.
- Le développement d'un pilote du périphérique sous Linux embarqué en mode utilisateur *(user mode driver*).
- Le développement de l'application de test du périphérique sous Linux embarqué.
- La mesure de temps de latence du noyau standard et du noyau Xenomai.

Nous allons voir dans un premier temps la mise en œuvre de Linux embarqué sur la carte cible ZedBoard.

14.2. Carte cible Digilent ZedBoard

La carte ZedBoard est une carte d'évaluation développée par la société Digilent et met en œuvre le circuit FPGA Zynq Z-7020 d'AMD.

Le circuit FPGA Zynq Z-7020 permet de réaliser un système SoPC et est constitué de logique programmable couplée à un processeur *hardcore* double coeur ARM Cortex-A9.

Dans un circuit FPGA Zynq, on appelle PS (*Processing System*) la partie processeur et ses périphériques associés.

La partie PS inclut :

- Les deux cœurs ARM Cortex-A9.
- Le bus AMBA-AXI (*Advanced Microcontroller Bus Architecture-Advanced eXtensible Interface).*
- les E/S GPIO et les liens série I2C, UART, CAN et SPI.
- Le contrôleur des mémoires QuadSPI, NAND et NOR.
- Le contrôleur de mémoire vive DDR3.

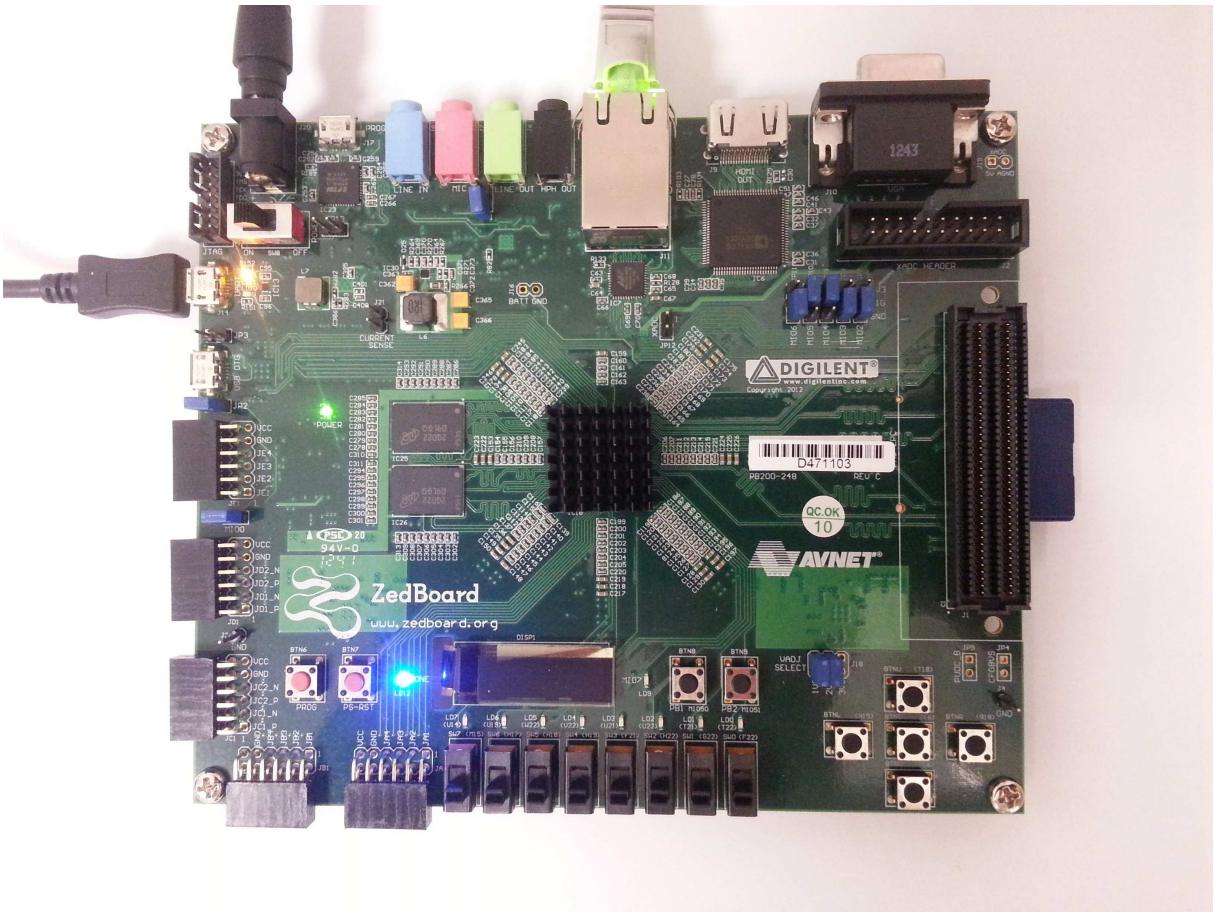
La partie PL (*Programmable Logic*) correspond à la partie logique programmable.

Le circuit FPGA utilisé par la carte ZedBoard est le circuit AMD XC7Z020-CLG484 (*Zynq-7020 AP SOC*). Le circuit Zynq Z-7020 est un circuit milieu de gamme dans la série et reprend la même logique programmable que les circuits AMD Artix. Les circuits Zynq haut de gamme Z-7030 et Z-7045 reprennent quant à eux la logique programmable des circuits AMD Kintex.

La carte Zedboard possède ainsi les éléments suivants :

- Processeur ARM Cortex-A9 à 533 MHz avec 32 Ko de cache L1 et 512 Ko de cache externe L2.
- 512 Mo de RAM DDR3.
- 32 Mo de mémoire Flash Quad SPI.
- Slot mémoire SD.
- Sorties HDMI (audio et vidéo).
- Sortie VGA.
- Ethernet (10/100/1000 Mb/s).
- Port de *débug* ARM DAP (*Debug Access Port*).
- Port USB 2.0 OTG *device* et *host*.
- Port USB-JTAG.
- Port USB-UART.
- Connecteur FMC (*FPGA Mezzanine Connector*).
- 5 ports d'extension Pmod.
- 9 leds utilisateur.
- 8 switchs.
- 7 boutons poussoir.

L'image suivante présente la carte cible ZedBoard :



Carte cible ZedBoard

Le choix du périphérique de *boot* de la carte ZedBoard se fait par la configuration de 3 *jumpers* suivant le tableau suivant :

	MIO[5]	MIO[4]	MIO[3]
JTAG	0	0	0
NOR	0	0	1
NAND	0	1	0
QSPI	1	0	0
SD	1	1	0

Configuration des *jumpers* MIO[3] à MIO[5] pour le choix du mode de boot

Nous utiliserons pour la suite le *boot* depuis la carte SD, ce qui permettra d'utiliser le *bootloader u-boot*.

15. EX 1 AMD : GENERATION DU RAM DISK ET INTEGRATION D'UNE APPLICATION

Nous allons voir comment compiler le noyau Linux embarqué avec son *RAM disk* exécuté par le processeur Cortex-A9 de la carte cible ZedBoard.

Par la suite, on adoptera les conventions suivantes :
Commande Linux PC hôte pour le développement croisé :

```
host% commande Linux
```

Commande Linux PC hôte pour le développement AMD :

```
[Xilinx EDK]$
```

Commande Linux embarqué sur la carte cible ZedBoard :

```
ZedBoard:# commande Linux embarqué
```

Commande *u-boot* sur la carte cible ZedBoard :

```
U-Boot> commande u-boot
```

Nous allons voir comment rajouter une application dans le *RAM disk* utilisé par le noyau Linux standard exécuté par le processeur Cortex-A9 de la carte cible ZedBoard. Nous allons dans un premier générer notre propre *RAM disk*.

- Démarrer le PC sous Linux. Se connecter sous le nom **se01**, mot de passe : **se01** ☺ pour le groupe 1 et sous le nom **se02**, mot de passe : **se02** ☺ pour le groupe 2.

- Se placer dans son répertoire de travail :
  `host% cd`

- Se connecter à la carte ZedBoard (cible) en utilisant l'outil `minicom` :
  `host% minicom -b 115200 -D /dev/ttyACM0`
  Pour sortir de `minicom`, il suffit de taper la combinaison de touches : CTRL A, Z pour accéder au menu et taper q pour quitter. On arrêtera le compte à rebours de 3 secondes d'*u-boot* en appuyant sur la touche espace du clavier…

- Recopier le fichier `tp-ZedBoard.tgz` sous `/home/kadionik/` :
  `host% cp /home/kadionik/tp-ZedBoard.tgz .`

- Décompresser et installer le fichier `tp-ZedBoard.tgz` :
  `host% tar -xvzf tp-ZedBoard.tgz`

- Se placer ensuite dans le répertoire `ZedBoard/`. **L'ensemble du travail sera réalisé à partir de ce répertoire ! Les chemins seront donnés par la suite en relatif par rapport à ce répertoire...**
  `host% cd ZedBoard`

- Créer le système de fichiers *root* squelette `root_fs` pour la carte cible ZedBoard :
```
host% cd ramdisk
host% ./goskel
```

- Compiler `busybox` :
```
host% cd ramdisk
host% cd busybox
host% ./go
```

- Générer le système de fichiers *root* final `root_fs` pour la carte cible ZedBoard. Il est demandé à un moment donné de rentrer son mot de passe (`[sudo] password for se01/se02 :`):
```
host% cd ramdisk
host% ./gorootfs
```

- Se placer dans le répertoire `tst/hello/` et modifier le fichier `hello.c` afin de créer le fameux « *Hello World* » :
```
host% cd tst/hello
host% gedit hello.c
```

- Compiler l'application `hello` pour la carte cible ZedBoard :
```
host% ./go
```

- Installer l'application `hello` dans le système de fichiers *root* qui servira de base au *RAM disk* :
```
host% ./goinstall
```

- Se placer dans le répertoire `rootfs/` pour regénérer le *RAM disk*. Le système de fichiers *root* est sous `root_fs/` et notre application `hello` a été précédemment copiée sous `root_fs/bin/` :
```
host% cd ramdisk
host% ls root_fs/bin
```

- Générer le *RAM disk*. Il est demandé à un moment donné de rentrer son mot de passe (`[sudo] password for se01/se02 :`). Que fait le *shell script* `goramdisk` ? Comment s'appelle le fichier *RAM disk* ?
```
host% cd ramdisk
host% sudo ./goramdisk
```

- Installer le nouveau *RAM disk* dans le répertoire de téléchargement d'*u-boot* `/tftpboot` :
```
host% ./goinstall
```

Astuce !

Pour éviter de regénérer à chaque fois le *RAM Disk* et recharger le noyau Linux, on fera une compilation croisée puis on recopiera sous `/tftpboot` l'exécutable ainsi produit :

```
host% ./go
host% cp mon_appli_exe /tftpboot
```

ou plus simplement avec les *shell scripts* fournis :

```
host% ./go
host% ./goinstall
```

On pourra alors télécharger par TFTP l'exécutable en utilisant la commande `tftp` et lancer l'application :

```
target# tftp -g -r mon_appli_exe @IP_host
target# chmod u+x mon_appli_exe
target# ./mon_appli_exe
```

16. EX 2 AMD : MISE EN ŒUVRE DE LINUX EMBARQUE SUR LA CARTE CIBLE

Nous allons voir comment compiler le noyau Linux embarqué exécuté par le processeur Zynq de la carte cible ZedBoard.

- Compiler le noyau Linux standard pour la carte cible ZedBoard :
```
host% cd linux
host% ./go
```

- Installer le fichier du noyau Linux dans le répertoire de téléchargement d'*u-boot* `/tftpboot` :
```
host% ./goinstall
```

- Recharger (optionnel) le *design* de référence `system.bit` dans le circuit FPGA de la carte cible ZedBoard. Que fait le *shell script* `load-design` ?
```
host% cd ZedBoard
host% mbsdk2
[Xilinx EDK]$ ./load-design
```

- Depuis *u-boot* de la carte cible ZedBoard, lancer la commande suivante. Quels sont les 3 fichiers téléchargés depuis le PC hôte en RAM de la carte ZedBoard et quel est leur rôle ?
```
U-Boot> run ramboot
```

- Observer les traces de boot du noyau Linux standard dans la fenêtre `minicom` :

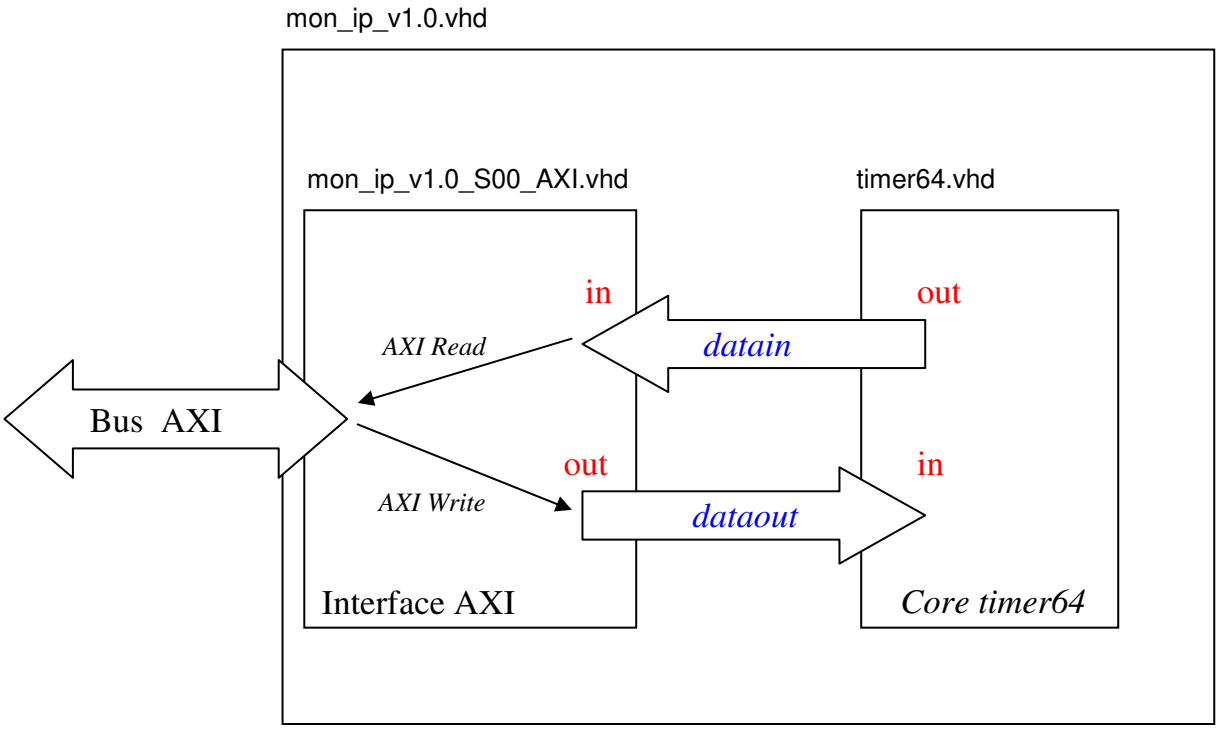
```
Starting kernel ...

Booting Linux on physical CPU 0x0
Linux version 6.1.0-xilinx-48445-gf9c8e14ae03c (kadionik@ipcchipik) (arm-
buildroot-linux-gnueabihf-gcc.br_real (Buildroot 2021.11-4428-g6b
6741b) 11.3.0, GNU ld (GNU Binutils) 2.38) #16 SMP PREEMPT Tue Mar 14
16:29:43 CET 2023
CPU: ARMv7 Processor [413fc090] revision 0 (ARMv7), cr=18c5387d
CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing instruction cache
OF: fdt: Machine model: xlnx,zynq-7000
earlycon: cdns0 at MMIO 0xe0001000 (options '115200n8')
printk: bootconsole [cdns0] enabled
Memory policy: Data cache writealloc
cma: Reserved 16 MiB at 0x1f000000
percpu: Embedded 15 pages/cpu s31872 r8192 d21376 u61440
Built 1 zonelists, mobility grouping on.  Total pages: 130048
Kernel command line: console=ttyPS0,115200 root=/dev/ram
ramdisk_size=131072 rw
earlyprintk earlycon
. . .
clocksource: jiffies: mask: 0xffffffff max_cycles: 0xffffffff, max_idle_ns:
1911
2604462750000 ns
futex hash table entries: 512 (order: 3, 32768 bytes, linear)
pinctrl core: initialized pinctrl subsystem
NET: Registered protocol family 16
DMA: preallocated 256 KiB pool for atomic coherent allocations
cpuidle: using governor menu
hw-breakpoint: found 5 (+1 reserved) breakpoint and 1 watchpoint registers.
hw-breakpoint: maximum watchpoint size is 4 bytes.
```

```
zynq-ocm f800c000.ocmc: ZYNQ OCM pool: 256 KiB @ 0x(ptrval)
e0001000.serial: ttyPS0 at MMIO 0xe0001000 (irq = 24, base_baud = 3125000)
is a
xuartps
printk: console [ttyPS0] enabled
printk: console [ttyPS0] enabled
printk: bootconsole [cdns0] disabled
printk: bootconsole [cdns0] disabled
vgaarb: loaded
. . .
Xilinx Zynq CpuIdle Driver started
sdhci: Secure Digital Host Controller Interface driver
sdhci: Copyright(c) Pierre Ossman
sdhci-pltfm: SDHCI platform and OF driver helper
mmc0: SDHCI controller on e0100000.sdhci [e0100000.sdhci] using ADMA
ledtrig-cpu: registered to indicate activity on CPUs
clocksource: ttc_clocksource: mask: 0xffff max_cycles: 0xffff, max_idle_ns:
5375
38477 ns
. . .
RAMDISK: gzip image found at block 0
mmc0: new high speed SDHC card at address 0007
mmcblk0: mmc0:0007 SD04G 3.71 GiB
 mmcblk0: p1
EXT4-fs (ram0): mounted filesystem with ordered data mode. Opts: (null)
VFS: Mounted root (ext4 filesystem) on device 1:0.
devtmpfs: mounted
Freeing unused kernel memory: 1024K
Run /sbin/init as init process
    Hostname       : ZedBoard
    Kernel release : Linux 6.1.0-xilinx-48445-gf9c8e14ae03c
    Kernel version : #16 SMP PREEMPT Tue Mar 14 16:29:43 CET 2023

 Mounting /proc             : [SUCCESS]
 Mounting /sys              : [SUCCESS]
 Mounting /dev              : [SUCCESS]
 Mounting /dev/pts          : [SUCCESS]
 Enabling hot-plug          : [SUCCESS]
 Populating /dev            : [SUCCESS]
 Mounting other filesystems : [SUCCESS]
 Starting telnetd           : [SUCCESS]
 Network configuration      : [SUCCESS]

System initialization complete.

Please press Enter to activate this console.

ZedBoard:/# uname -r
6.1.0-xilinx-48445-gf9c8e14ae03c
ZedBoard:/#
```

17. EX 3 AMD : INTEGRATION D'UN PERIPHERIQUE MATERIEL LIBRE

17.1. Introduction

L'objectif de ce TP est d'intégrer un périphérique matériel Libre, en l'occurrence un *timer* 64 bits utilisé dans sa fonctionnalité de comptage qui pourra communiquer avec le microprocesseur *hardcore* Cortex-A9 du circuit FPGA Zynq de la carte cible via le bus AXI (*Advanced eXtensible Interface*) du processeur.



AXI Write : dataoutX <= slv_regX; X∈ [0,3]
AXI Read : reg_data_out <= datainX;

Timer 64 bits et son interfaçage sur le bus AXI

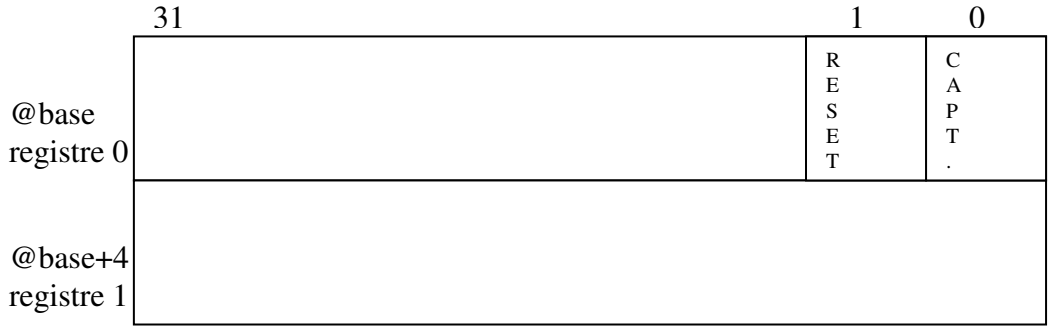
Ce *timer* 64 bits conçu par nos soins s'interface directement sur le bus AXI du processeur Cortex-A9 du circuit FPGA Zynq. Il a été grandement simplifié et ne propose que la fonction de comptage du nombre de périodes d'horloge du circuit FPGA depuis son reset. La fréquence est de 100 MHz soit 10 ns de période. Il est donc possible de faire des mesures de temps avec une précision de 10 ns avec ce *timer*… Une seconde de temps écoulée correspond ainsi à l'augmentation de la valeur courante du compteur de 100 millions…

Les fichiers sources VHDL du *timer* sont directement fournis pour gagner du temps mis à part un ajout simple à faire dans le fichier source.

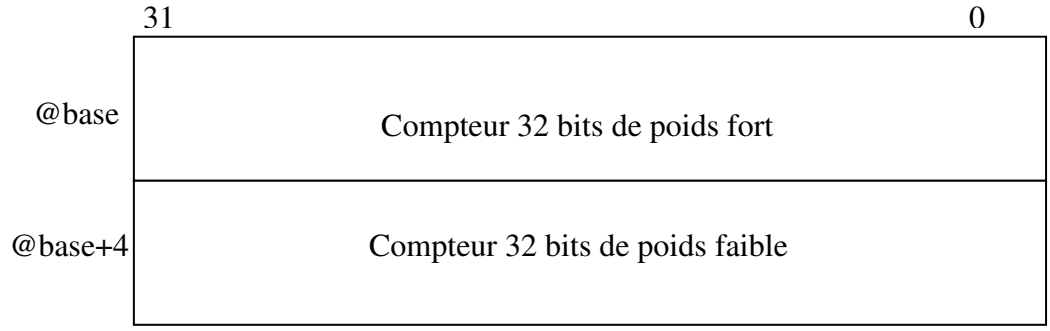
Ce *timer* possède des registres de contrôle et de données mappés dans l'espace d'adressage du processeur et directement accessibles par le bus AXI :

- Un registre de capture sur 64 bits à concaténer par 2 **lectures** de 32 bits (le bus AXI est un bus 32 bits) aux adresses base et base+4.
- Un registre de contrôle accessible en **écriture seulement** seulement à l'adresse de base :
  - Ecriture de 1 dans le bit 0 du registre : on enregistre la valeur courante du compteur et on la transfère vers le registre de capture du *timer*.
  - Ecriture de 1 dans le bit 1 du registre : on réinitialise à 0 le compteur du *timer*.

Nous pouvons résumer cela par la figure suivante :



Accès registres en écriture



Accès registres en lecture

Registres de contrôle et de données du *timer* 64 bits

Le *timer* fonctionne met en œuvre 3 fichiers sources VHDL :

- Fichier `mon_ip_v1.0_S00_AXI.vhd` : ce fichier VHDL correspond à l'implantation de l'interface esclave AXI. Il est généré par Vivado.
- Fichier `timer64.vhd` : ce fichier VHDL est le compteur 64 bits en lui-même.
- Fichier `mon_ip_v1.0.vhd` : ce fichier VHDL correspond au bloc IP *timer64* incluant les 2 fichiers précédent. Il est généré par Vivado.

Le fichier source `timer64.vhd` est le suivant :

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity timer64 is
  port(
    clk, reset: in std_logic;
    start: in std_logic;
    capture_high: out std_logic_vector(31 downto 0);
    capture_low: out std_logic_vector(31 downto 0)
    );
end timer64;

architecture arch of timer64 is
  signal timer64: unsigned((2*32)-1 downto 0);
  signal capture_register: unsigned((2*32)-1 downto 0);
begin


  core: process(clk, reset)
  begin
    if (reset = '1') then
      timer64 <= (others=>'0'); -- Asynchrone
    elsif (clk'event and clk='1') then
       -- A compléter . . .
    end if;
  end process core;

  capture: process(clk, start)
  begin
    if (clk'event and clk='1') then
      if (start = '1') then
        capture_register <= timer64;
      end if;
    end if;
  end process capture;

-- output
-- Compteur 32 bits poids fort @
  capture_high <= std_logic_vector(capture_register((2*32)-1 downto 32));

-- Compteur 32 bits poids faible @+4
  capture_low <= std_logic_vector(capture_register(32-1 downto 0));
end arch;
```

Fichier source `timer64.vhd` du *timer* 64 bits

Le fichier est incomplet et est à compléter dans le processus VHDL *core*. L'entité/architecture s'appelle `timer64(arch)`.

On peut noter que :
- Le signal `reset` réinitialise le compteur à 0.
- Le signal `start` lance un *snaphot* de la valeur courante du compteur 64 bits.
- Les signaux `capture_high et capture_low` capturent les 32 bits de poids fort et les 32 bits de poids faible du *snapshot*.

Si l'on regarde le fichier `mon_ip_v1.0.vhd` vers la ligne 151, on a le code VHDL suivant :

```
clk <= s00_axi_aclk;
reset <= dataout0(1);

-- Start capture
start <= dataout0(0);


-- Instantiate core
core: entity work.timer64(arch)
      port map(
         clk=>clk, reset=>reset, start=>start,
         capture_high=>datain0, capture_low=>datain1
      );
```

Extrait du fichier source `mon_ip_v1.0.vhd` du *timer* 64 bits

Le signal `reset` est bien relié au bit 1 du registre @base (`dataout0`) alors que le signal `start` (capture) est bien relié au bit 0 du registre @base (`dataout0`) du périphérique *timer* 64 bits.
De même, on lit bien le *snapshot* via le registre @base (`datain0`) pour les 32 bits de poids fort et le registre @base+4 (`datain1`) pour les 32 bits de poids fort du périphérique *timer* 64 bits.

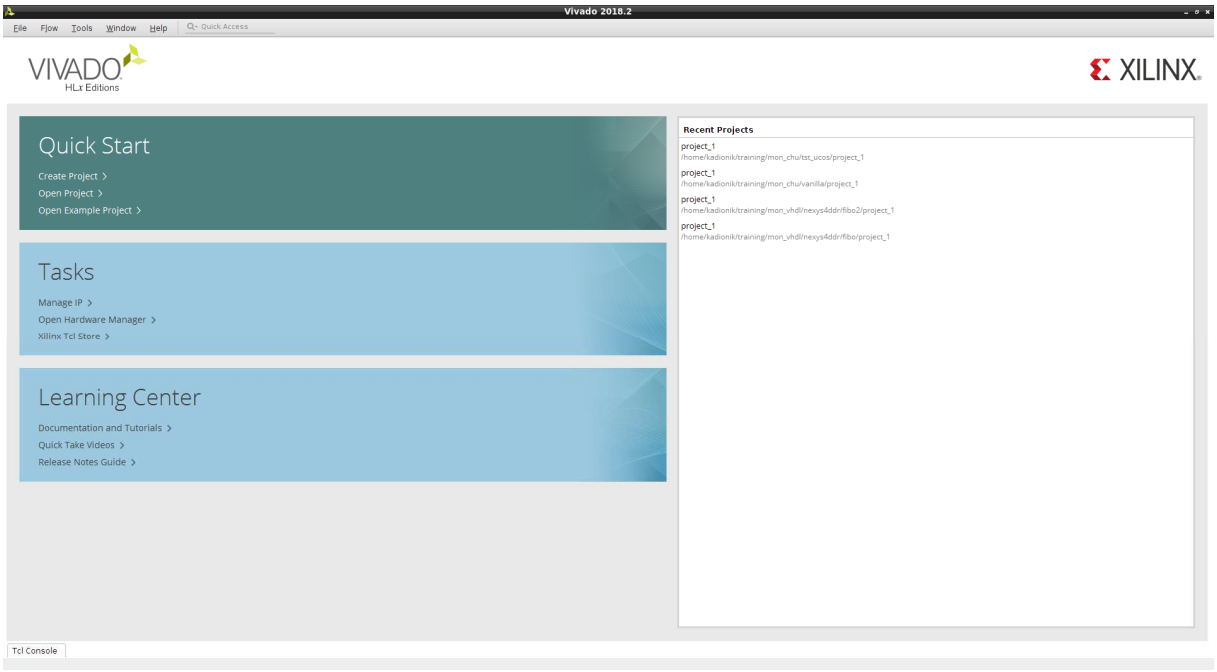
17.2. Intégration d'un périphérique. *Timer* 64 bits

- Se placer dans le répertoire `ZedBoard/` :
  ```
  host% cd ZedBoard
  ```

- Se placer ensuite dans le répertoire `design/design_ZedBoard/` et l'on se placera dans l'environnement AMD avec le *shell script* `mbsdk2` :
  ```
  host% cd design/design_ZedBoard/
  host% mbsdk2
  ```

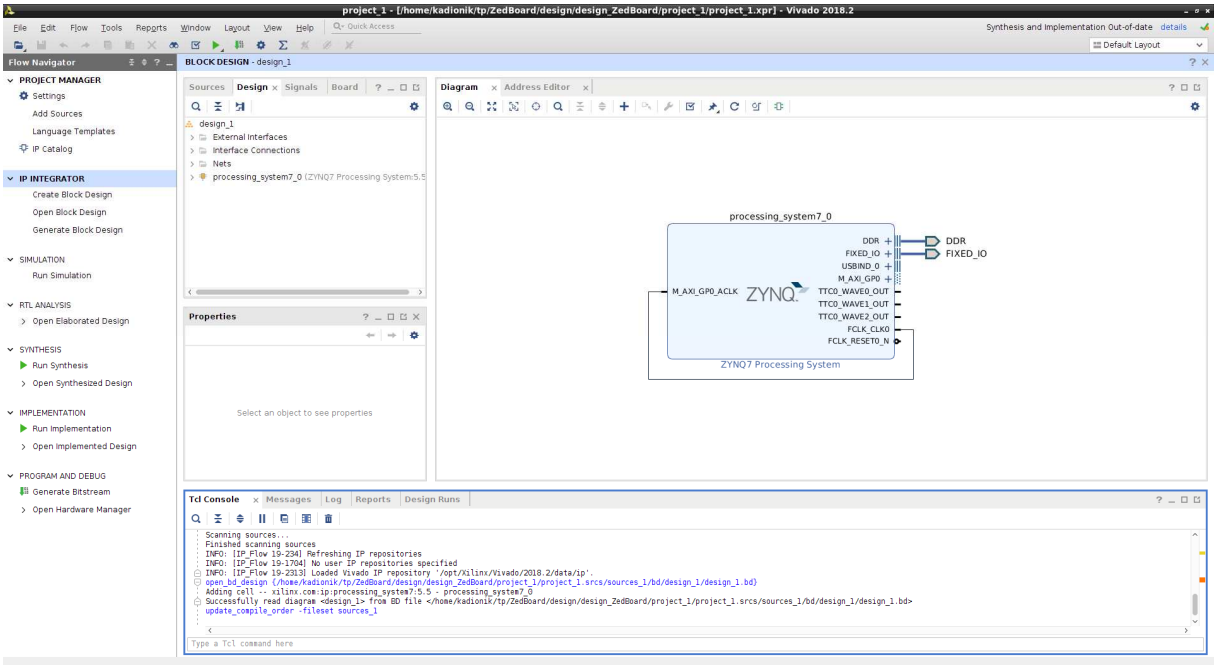
- Lancer l'outil AMD Vivado de création de SoPC :
  ```
  [Xilinx EDK]$ vivado
  ```

- Ouvrir le projet Vivado `project_1.xpr` se trouvant dans le répertoire courant par :
  - *Open Project*.
  - Sélection de *project_1 > project_1.xpr*.



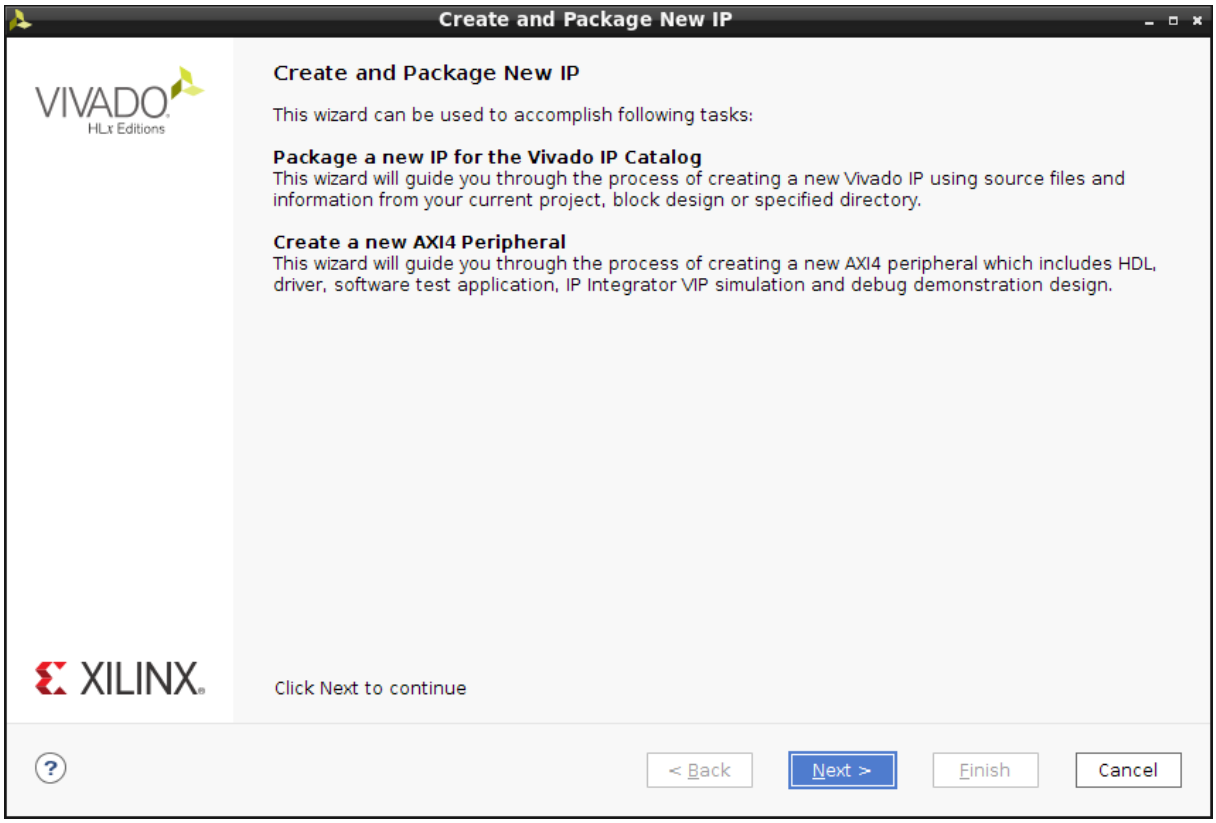
Projet SoPC `project_1.xpr`

- Ouvrir le *Block Design* :
  - *Flow Navigator > IP Integrator > Open Block Design.*

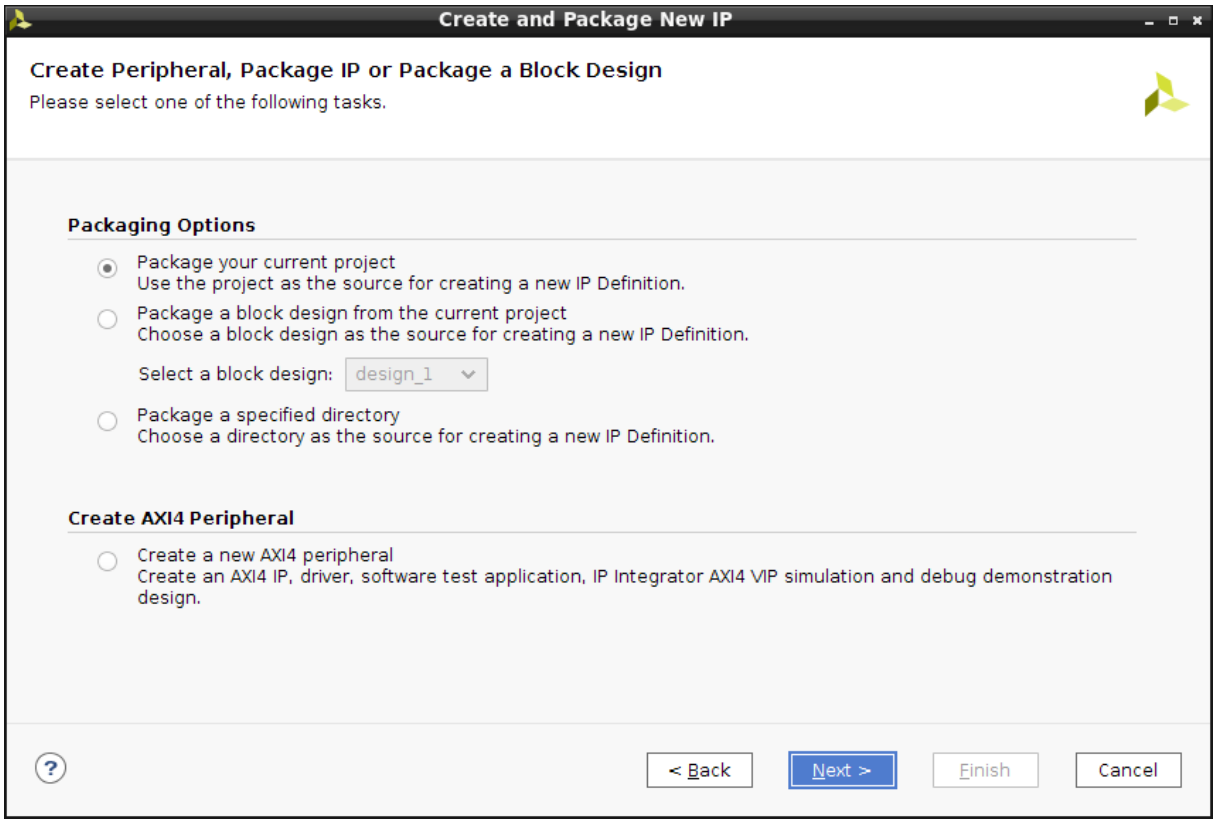


Ouverture du *Block Design*

- Créer un nouveau périphérique matériel. On suivra pas à pas les différentes recopies d'écran suivantes. Sauf indications contraires, on gardera les valeurs par défaut :
  - *Tools > Create and Package New IP.*

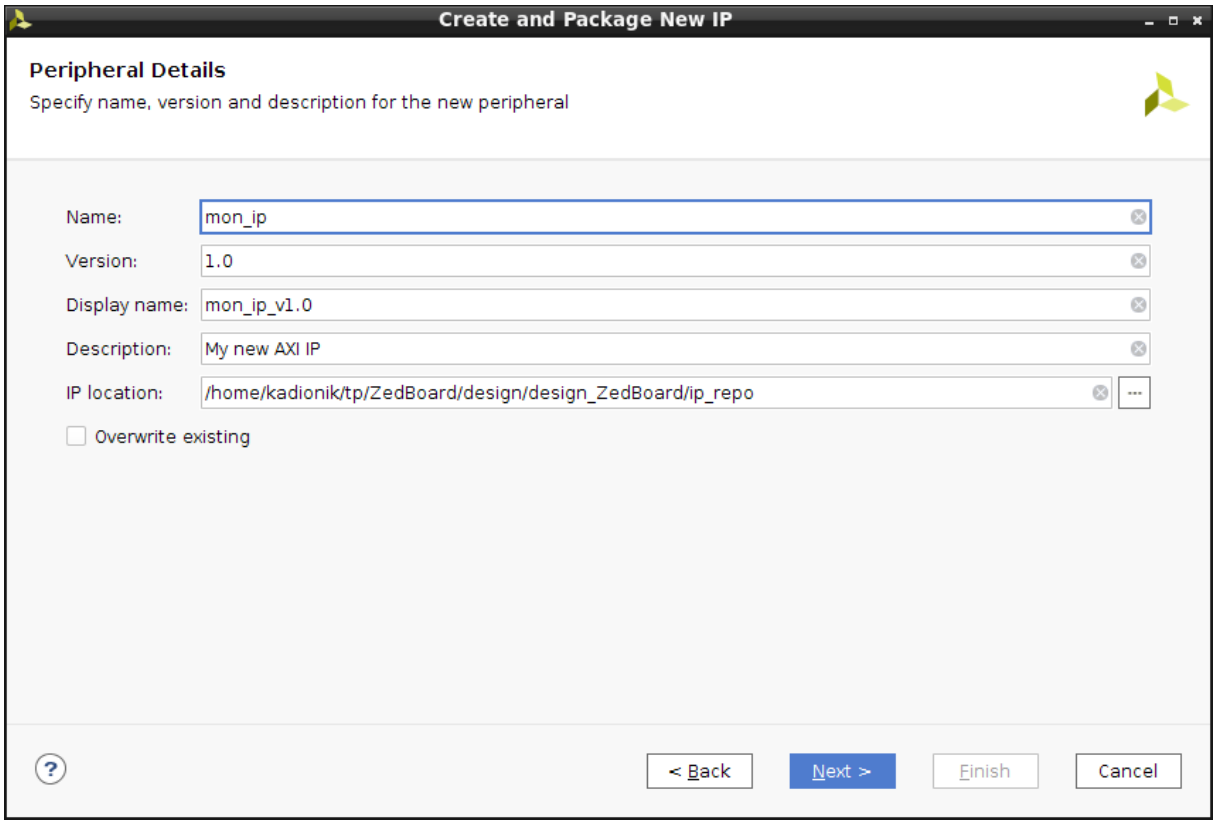


Création d'un périphérique (1)



Création d'un périphérique (2)

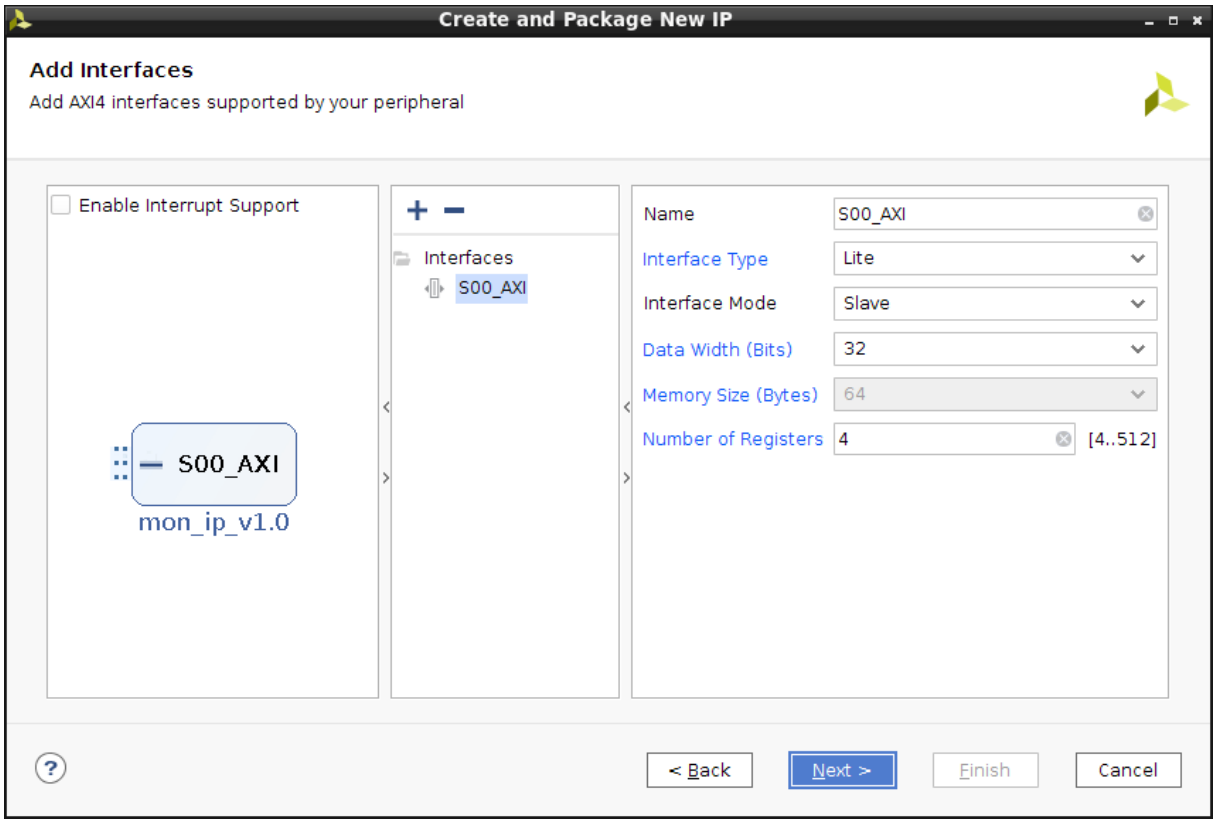
- Sélectionner :
    - *Create a new AXI4 Peripheral*.



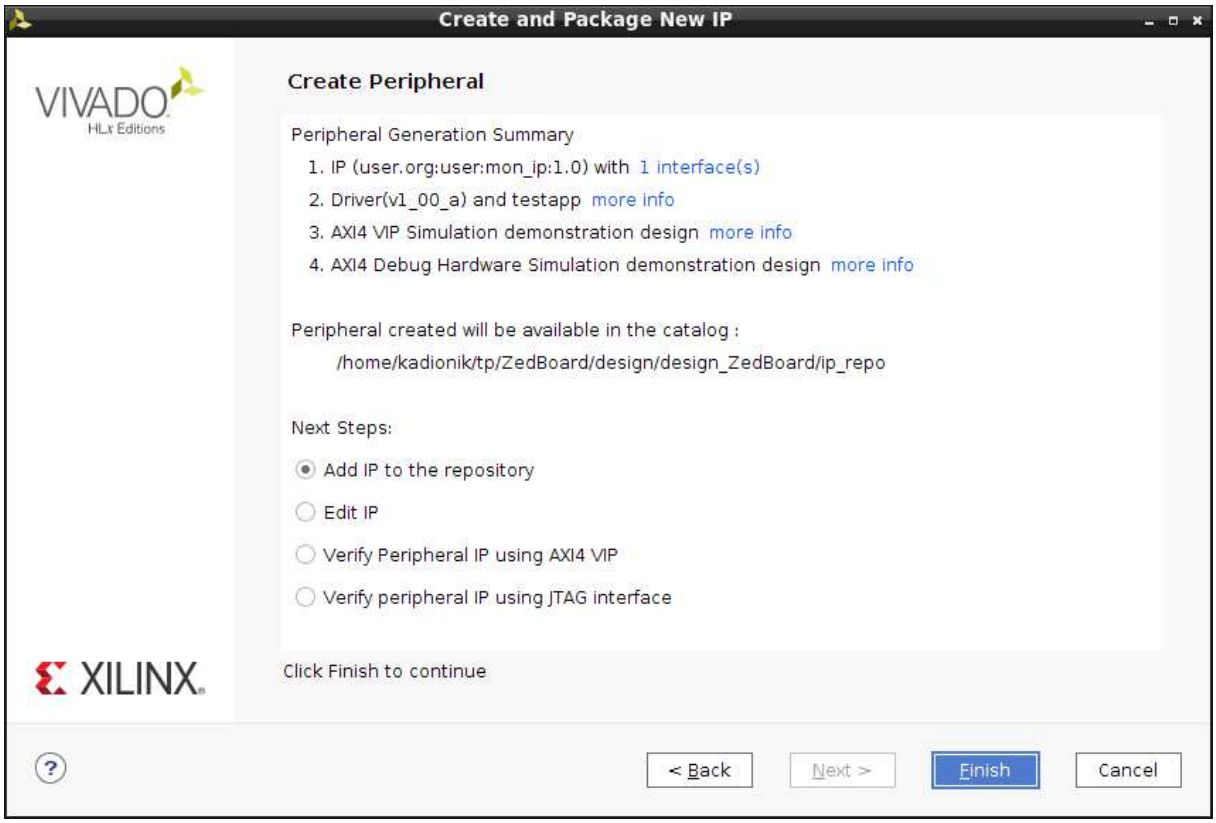
Création d'un périphérique (3)

On choisira les valeurs suivantes :

- *Name* : **mon_ip**.
- Le reste inchangé.



Création d'un périphérique (4)

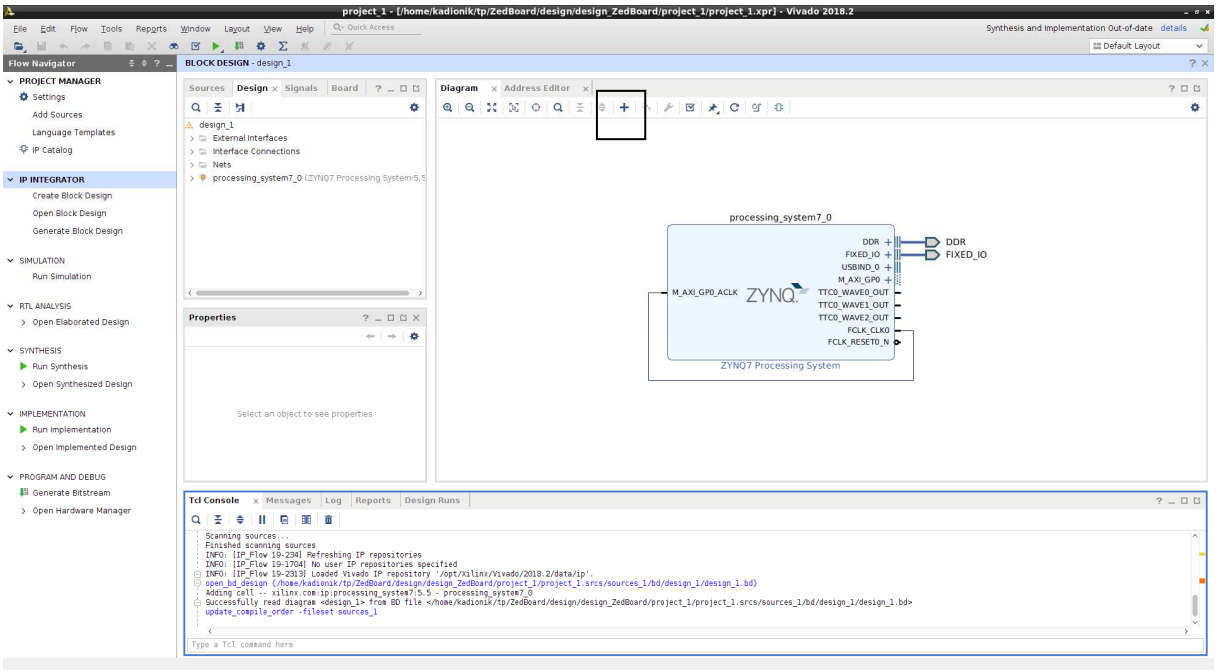


Création d'un périphérique (5)

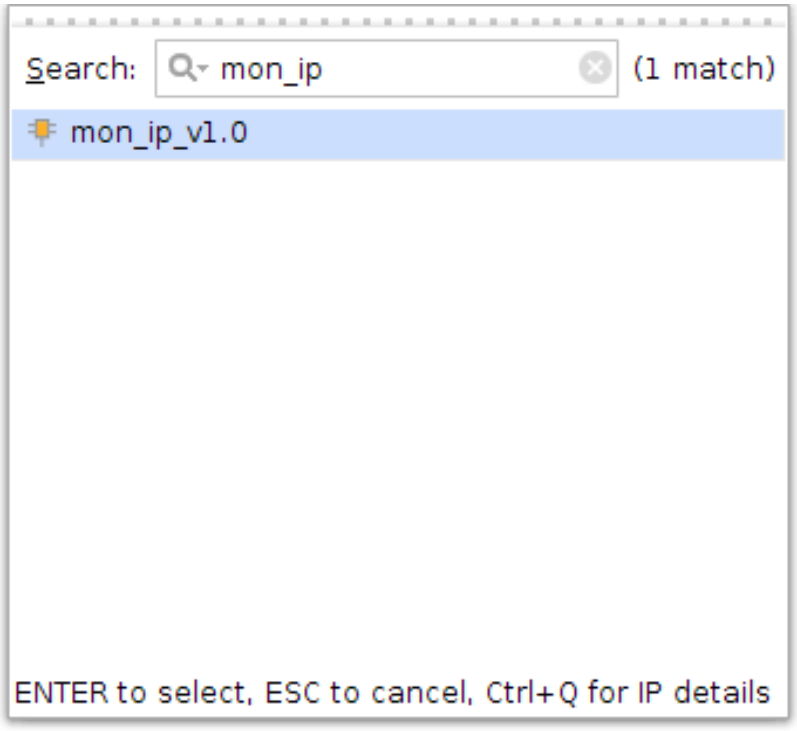
On choisira les valeurs suivantes :

- *Next Steps : Add IP to the repository*.
- Cliquer sur le bouton *Finish*.
- Le périphérique est créé mais ce n'est qu'une coquille vide !

- Dans la zone *Diagram*, cliquer sur le bouton *Add IP* et ajouter l'IP `mon_ip` :

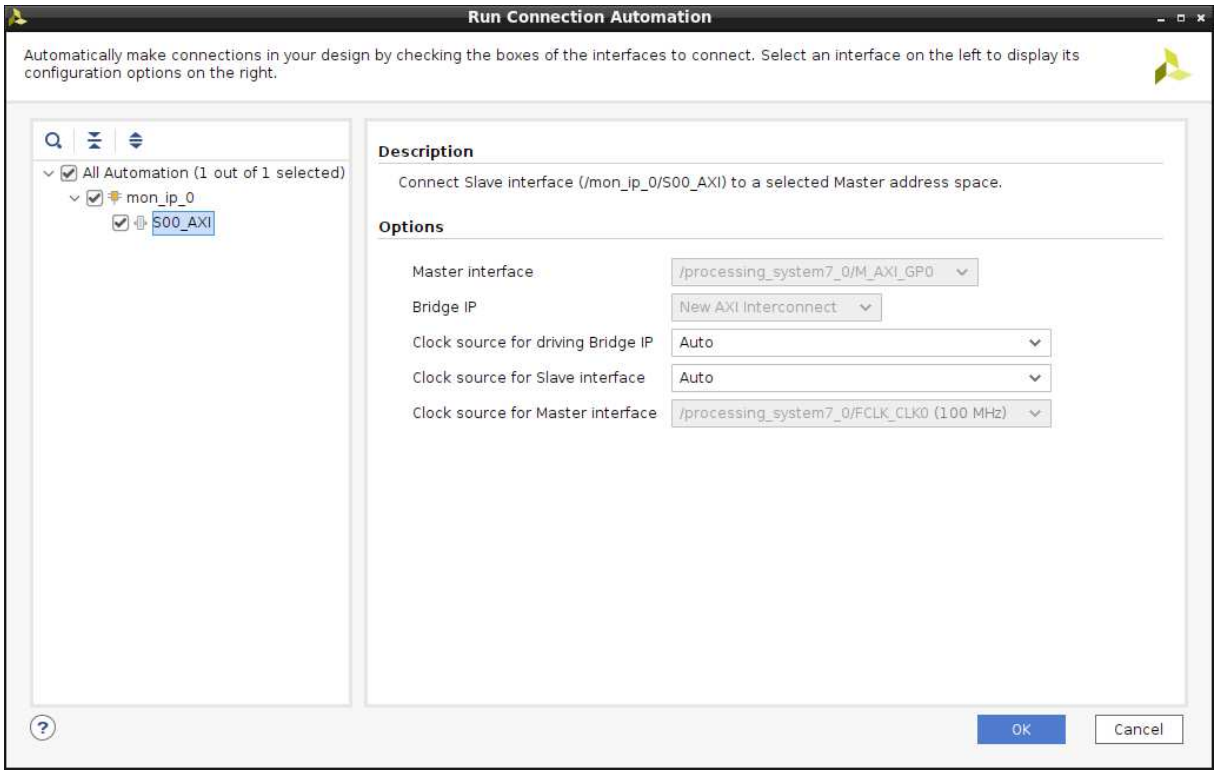


Ajout de l'IP `mon_ip` (6)



Ajout de l'IP `mon_ip` (7)

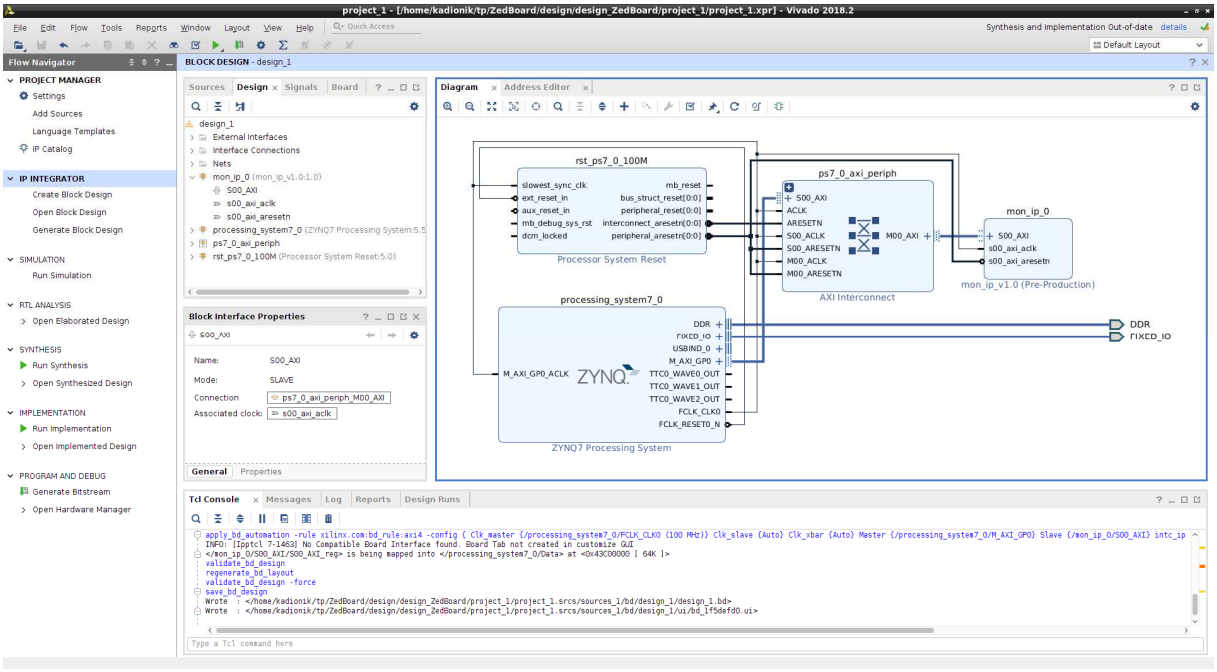
- Cliquer sur le texte *Run Connection Automation* en haut à droite dans la zone *Diagram* pour interfacer le bloc IP sur le bus AXI.



Ajout des connections du périphérique `mon_ip` au bus AXI (8)

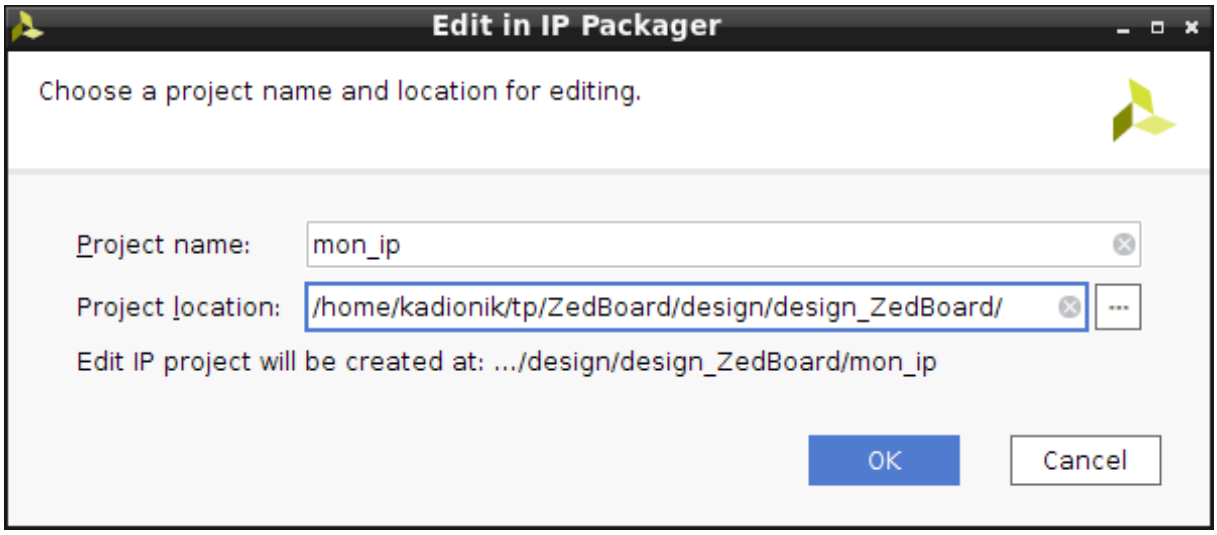
- Valider le *design* en appuyant sur la touche F6 et sauvegarder le *Block Design* (CTRL + S).

- On obtient alors le *design* `project_1.xpr` suivant :



Projet Vivado avec le bloc IP `mon_ip` (9)

- Dans la fenêtre *Diagram*, cliquer sur le bloc IP `mon_ip_0` et par appui sur la touche de droite de la souris (menu contextuel), choisir le menu *Edit in IP Packager* :



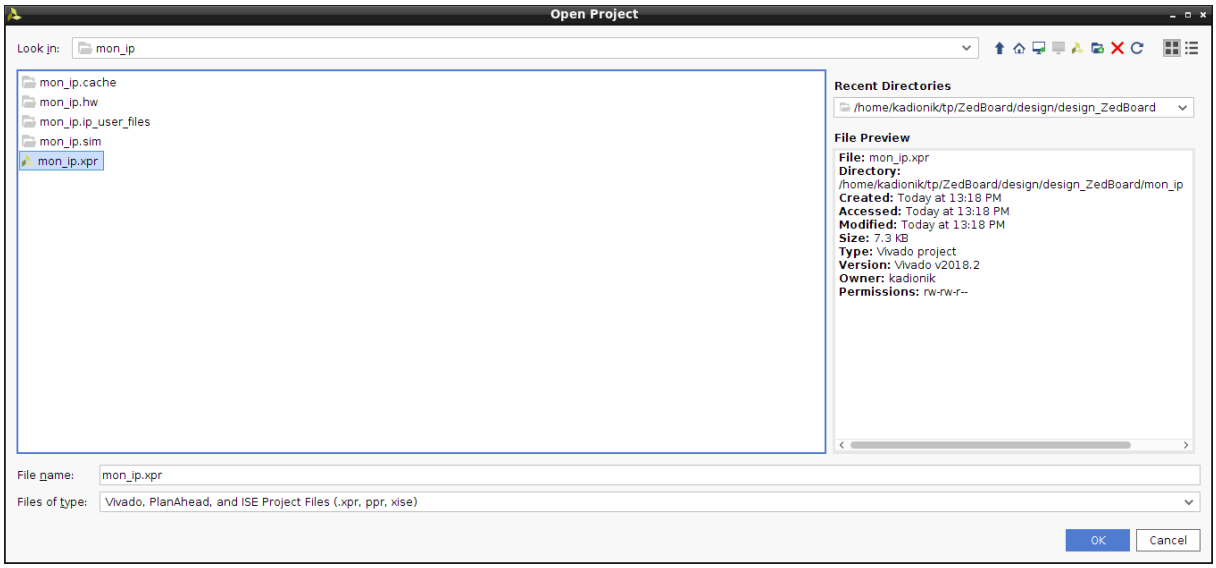
Création du projet Vivado mon_ip (10)

On choisira les valeurs suivantes :

- *Project name* : **mon_ip**.
- *Project Location* : **ZedBoard/design/design_ZedBoard**. Attention à bien respecter ce paramètre !!!
- Cliquer sur le bouton *OK*. *OK to overwrite* ? Yes !

- **Fermer aussitôt le projet Vivado ainsi créé !!!!**

- Ouvrir le projet Vivado `mon_ip.xpr` se trouvant dans le répertoire courant par :
  - *File > Open Project.*
  - Sélection de mon_ip > mon_ip.xpr.



Sélection du projet Vivado `mon_ip.xpr` (11)

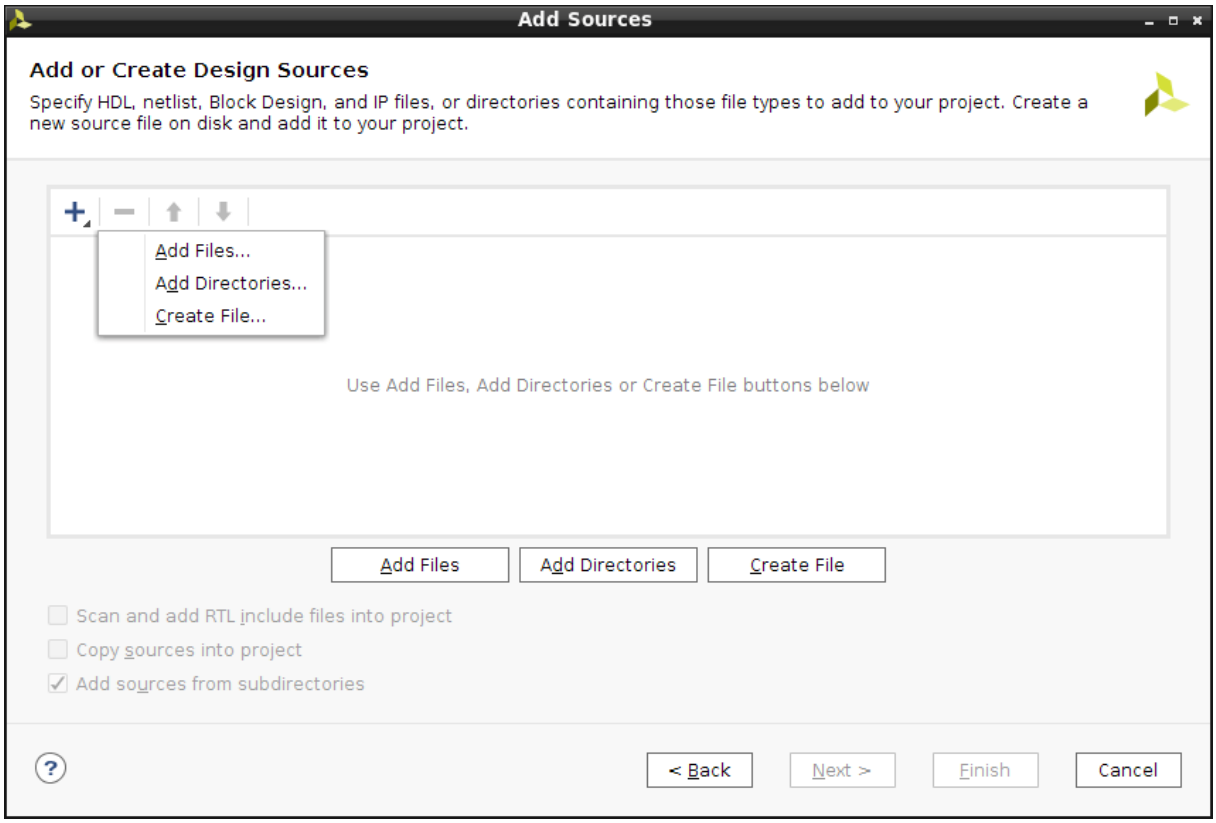
- Laisser le projet `project_1.xpr` ouvert.

- Remplacer les fichiers VHDL précédemment générés par l'ajout du périphérique avec ceux correspondant au *timer*. Le fichier `timer64.vdh` sera importé au projet puis modifié directement depuis Vivado :

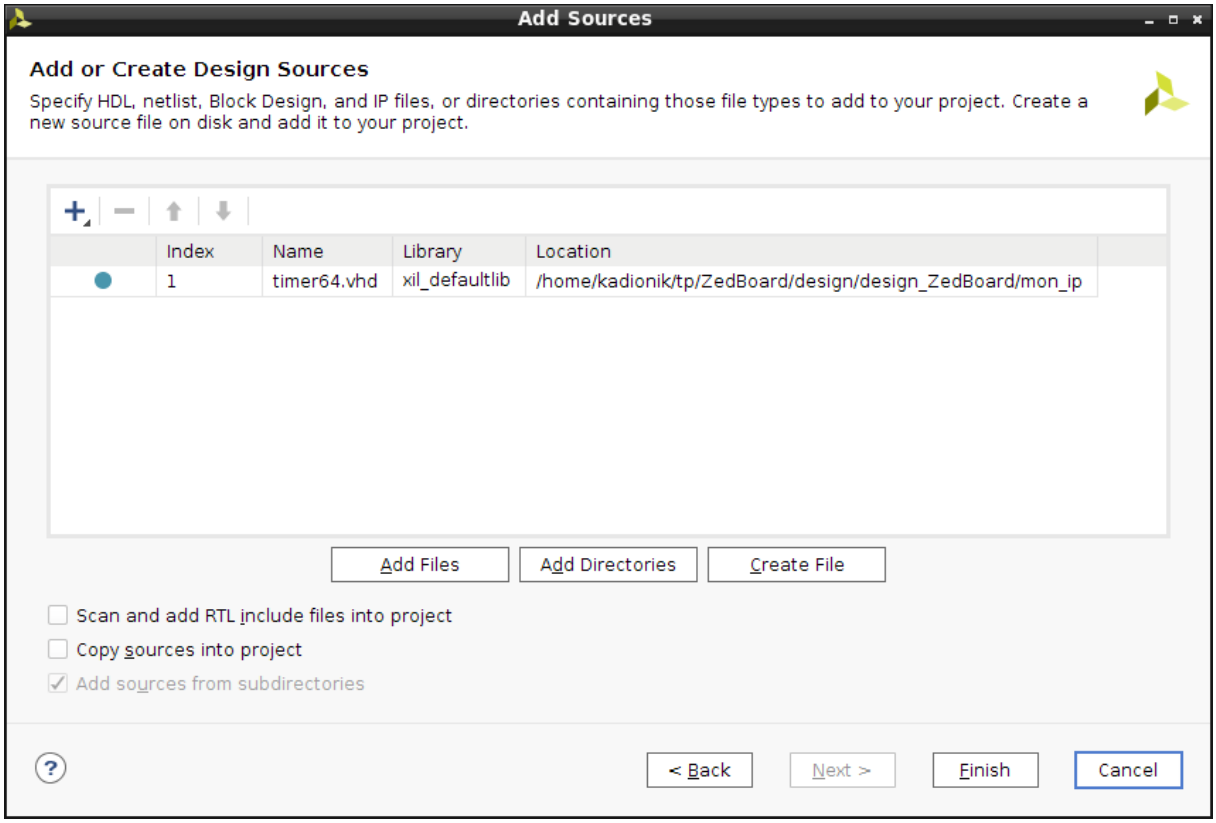
```
host% cd design/design_ZedBoard
host% cp ../ip/mon_ip_v1_0_S00_AXI.vhd ip_repo/mon_ip_1.0/hdl/
host% cp ../ip/mon_ip_v1_0.vhd ip_repo/mon_ip_1.0/hdl/
```

- On remarquera qu'il manque le fichier VHDL qui définit l'entité `timer64(arch).`

- Ajouter le fichier VHDL `timer64.vhd` :
  - *Flow Navigator > Project Manager > Add Sources.*
  - *Add or create design sources.*
  - *Add Files.*
  - Dans le répertoire `design/design_ZedBoard/mon_ip/`, choisir le fichier VDHL `timer64.vhd`.
  - Cliquer sur le bouton *Finish.*



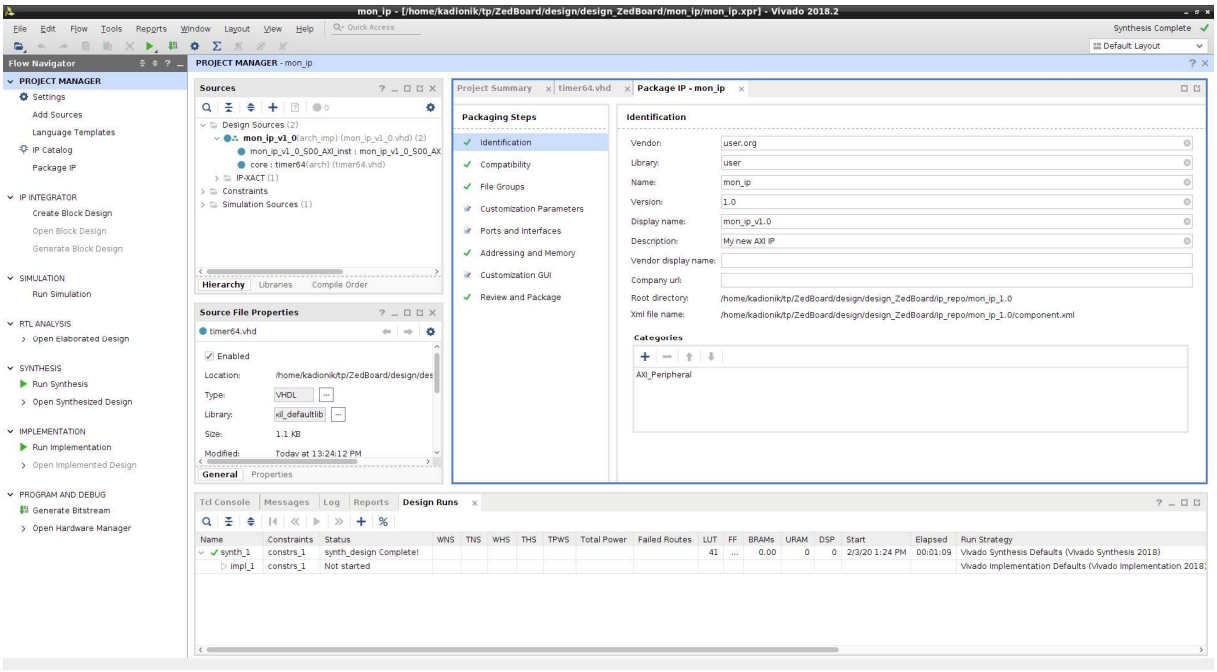
Ajout du fichier timer64.vhd au projet Vivado `mon_ip.xpr` (12)



Ajout du fichier timer64.vhd au projet Vivado `mon_ip.xpr` (13)

- Editer le fichier `timer64.vhd` et **compléter le code source à la ligne 25** pour ajouter la fonction d'incrémentation du compteur 64 bits.

- Lancer la synthèse du projet `mon_ip.xpr` pour vérification fonctionnelle par la commande :
    - *Flow Navigator > Synthesis > Run Synthesis.*



Ajout du fichier `timer64.vhd` au projet Vivado `mon_ip.xpr` (14)

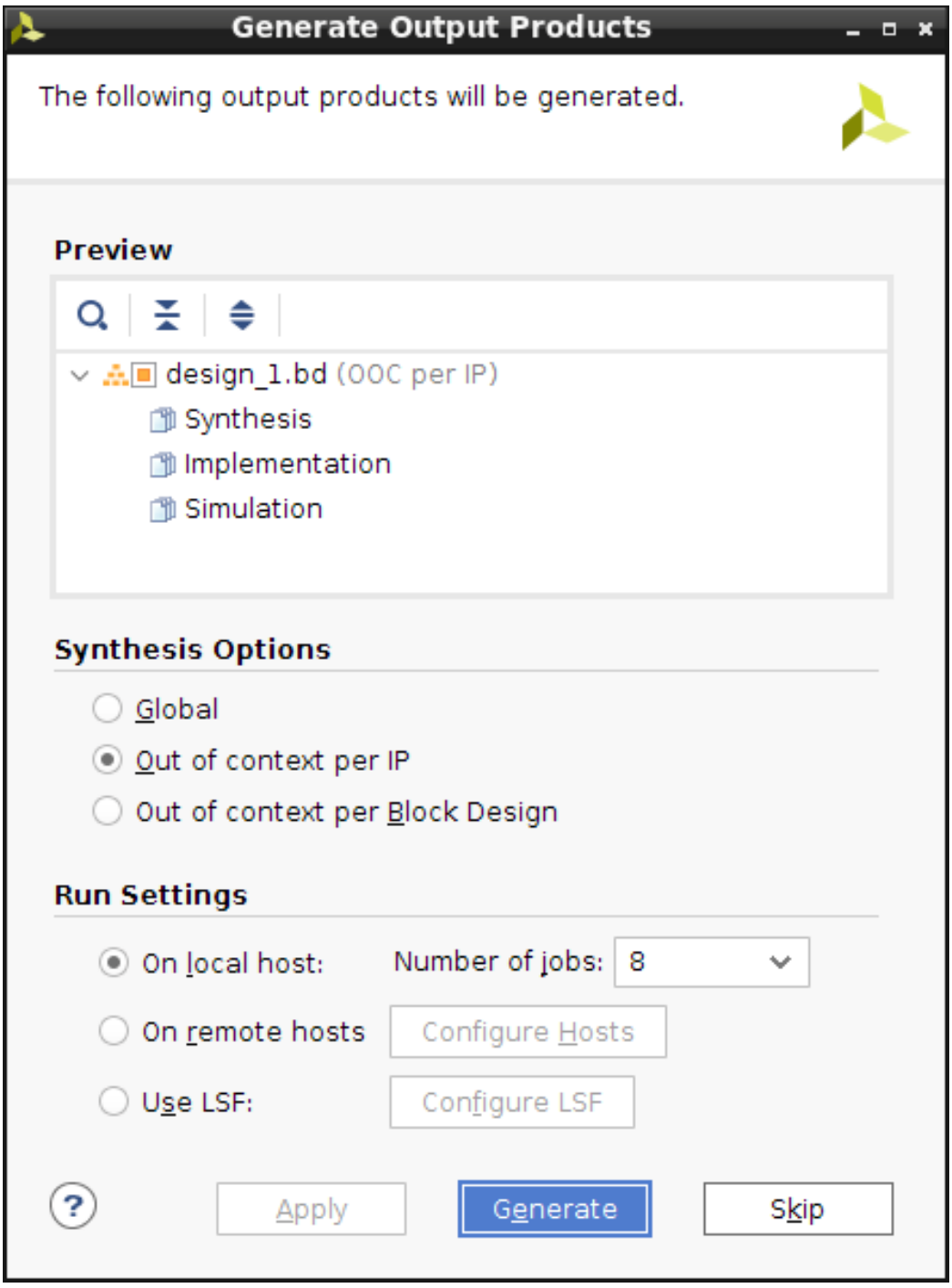
- Si la synthèse est OK, on *repackage* le bloc IP :
    - *Flow Navigator > Project Manager > Edit Packaged IP.*

- On passera en revue toutes les étapes de *Packaging Steps* :
    -
    - *Customization Parameters > Merge changes from Customization Parameters Wizard.*
    - *Review and Package > Re-Package IP.*

- La réalisation du périphérique *timer* est terminée. Fermer le projet :
    - *File > Close Project.*

- Revenir au projet Vivado `project_1.xpr`. On réalisera les actions dans l'ordre suivant :
    - Appuyer sur la touche F6 pour revalider le *design.*
    - Cliquer sur *Refresh IP Catalog.*
    - Cliquer sur *Upgrade Selected.*

- On regénère le *Block Design* :
  - *Flow Navigator > IP Integrator > Generate Block Design*.



Génération du *Block Design* (15)

- Lancer enfin la synthèse pour générer le fichier `.bit` :
    - *Flow Navigator > Program and Debug > Generate Bitstream.*

- Sortir de Vivado. Recopier le nouveau fichier de programmation `system.bit` à la place de l'ancien :

```
host% cd ZedBoard
host% mv system.bit system.bit.org
host% cp
design/design_ZedBoard/project_1/project_1.runs/impl_1/design_1
_wrapper.bit system.bit
```

- Programmer le fichier `system.bit` dans le circuit FPGA de la carte cible ZedBoard. **Il faudra faire cette opération à chaque reset de la carte cible** :

```
host% cd ZedBoard
host% mbsdk2
[Xilinx EDK]$ ./load-design
```

17.3. Tests logiciels

Pour tester le *timer*, il suffira d'interagir avec lui via son registre de contrôle et ses deux registres de données accessibles via le bus AXI et mappés dans l'espace d'adressage.
Pour les tests logiciels, on écrira un pilote de périphérique en mode utilisateur (*user mod driver*) qui évitera d'écrire un vrai pilote de périphérique. Un pilote de périphérique en mode utilisateur ouvre le périphérique `/dev/mem` en tant que superutilisateur et mappe une page mémoire de 4 Ko (ou plus) dans l'espace d'adressage du processus sur l'adresse de base des registres du périphérique via l'appel système `mmap()`. Dès lors, de simples lectures et écritures de 32 bits permettent d'interagir avec le périphérique…

❖ Test de la mesure de temps

- Se placer dans le répertoire `tst/tsttimer64/` :
```
host% cd tst/tsttimer64
```

- Analyser le fichier squelette `tsttimer64.c` donné en annexe et compléter le pour mesurer le delta toutes les secondes du compteur 64 bits. On notera que la constante `XPAR_MON_IP_0_S00_AXI_BASEADDR` définie dans le fichier `xparameters.h` correspond à l'adresse de base des registres du *timer* fixée sous Vivado :
```
host% gedit tsttimer64.c
```

- Compiler le fichier `tsttimer64.c` et installer l'application `tsttimer64` dans le système de fichiers *root* :
```
host% ./go
host% ./goinstall
```

- Regénérer le *RAM disk* :
```
host% cd ramdisk
host% sudo ./goramdisk
host% ./goinstall
```

- Relancer le noyau Linux depuis *u-boot* :
```
U-Boot> run ramboot
```

- Tester l'application `tsttimer64`. Quelle valeur théorique doit-on avoir en hexadécimal pour la valeur delta pour une seconde ? Pourquoi les valeurs delta mesurées diffèrent-elles de la valeur théorique ? Que se passe-t-il si l'on charge le système ?

❖ Test de l'incrémentation

- Se placer dans le répertoire `tst/tstinc/` :
  ```
  host% cd tst/tstinc
  ```

- Analyser le fichier squelette `tstinc.c` et compléter le pour réaliser un *snapshot* toutes les secondes du compteur 64 bits.

- Compiler le fichier `tstinc.c` et installer l'application `tstinc` dans le système de fichiers *root*.

- Regénérer le *RAM disk*.

- Relancer le noyau Linux depuis *u-boot* et tester l'application `tstinc`. Au bout de combien de temps aura-t-on le débordement du compteur 64 bits ?

18. EX 4 AMD : CREATION DU RAM DISK POUR LE NOYAU LINUX STANDARD

Nous allons regénérer le *RAM disk* qui sera utilisé par le noyau Linux standard exécuté par le processeur Cortex-A9 de la carte cible ZedBoard. Nous allons aussi y intégrer tous les utilitaires nécessaires pour tester les performances Temps Réel du noyau Linux standard.

- Se placer dans le répertoire `tst/jitter/` :
```
host% cd tst/jitter
```

- Analyser le fichier squelette `jitter.c` donné en annexe et compléter le pour réaliser la mesure en continu du *jitter*, c'est-à-dire la mesure de la plus grande différence entre la mesure mesurée de l'incrémentation du *timer* et son incrémentation théorique sur une seconde (valeur mesurée sur une seconde moins 100 millions en dizaines de ns) :

- Compiler le fichier `jitter.c` et installer l'application `jitter` dans le système de fichiers *root*.

- Générer les utilitaires de tests `cyclictest`, `stress`... Les installer dans le système de fichiers *root* :
```
host% cd tst
host% cd stress
host% ./go
host% ./goinstall
host% cd tst
host% cd schedutils
host% ./go
host% ./goinstall
host% cd tst
host% cd rt-tests
host% ./go
host% ./goinstall
```

- Regénérer le *RAM disk*.

- Télécharger le noyau Linux standard et son *RAM disk* dans la carte cible ZedBoard et tester l'application `jitter`.

19. EX 5 AMD : MESURE DE TEMPS DE LATENCE AVEC LE NOYAU LINUX STANDARD

Nous allons mesurer des temps de latence du noyau Linux standard dans le cas d'un noyau non stressé puis dans le cas d'un noyau stressé.

Pour stresser le noyau, on utilisera l'utilitaire `stress`.

Pour mesurer les temps de latence, on utilisera l'utilitaire `cyclictest`.

- Dévalider le *throttling* :
```
ZedBoard:# echo -1 > /proc/sys/kernel/sched_rt_runtime_us
```

Noyau standard non stressé :

- Lancer `jitter`. Noter le temps de latence maximum au bout de 5 minutes de tests :
```
ZedBoard:# jitter
```

- Lancer `cyclictest`. Noter le temps de latence maximum au bout de 5 minutes de tests :
```
ZedBoard:# cyclictest -n -p 99 -i 5000
```

Noyau standard stressé :

- Stresser le noyau avec `stress`. Que fait le programme `stress` ?
```
ZedBoard:# stress -c 50 -i 50 &
```

- Lancer `jitter`. Noter le temps de latence maximum au bout de 5 minutes de tests :
```
ZedBoard:# jitter
```

- Lancer `cyclictest`. Noter le temps de latence maximum au bout de 5 minutes de tests :
```
ZedBoard:# cyclictest -n -p 99 -i 5000
```

20. EX 6 AMD : CREATION DU RAM DISK POUR LE NOYAU LINUX XENOMAI

Nous allons voir comment créer le système de fichiers *root* qui sera utilisé par le noyau Linux Xenomai exécuté par le processeur de la carte cible ZedBoard. Nous allons aussi y intégrer tous les utilitaires nécessaires pour tester les performances Temps Réel du noyau Linux Xenomai.

- Créer le système de fichiers *root* squelette `root_fs` pour la carte cible ZedBoard :
```
host% cd ramdisk
host% ./goskel
host% ./gorootfs
```

- Générer les utilitaires de tests de Xenomai `cyclictest`, `latency`... :
```
host% cd xenomai
host% ./goconfig
host% ./go
host% ./goinstall
```

- Générer les utilitaires de tests `cyclictest`, `stress`...
```
host% cd tst
host% cd stress
host% ./go
host% ./goinstall
host% cd tst
host% cd schedutils
host% ./go
host% ./goinstall
host% cd tst
host% cd rt-tests
host% ./go
host% ./goinstall
```

- Se placer dans le répertoire `tst/hello_xenomai/` :
```
host% cd tst/hello_xenomai
```

- Analyser le fichier `hello_xenomai.c` donné en annexe qui est le fameux « *Hello World* » sous forme d'une tâche Xenomai périodique. Quelle API Xenomai a-t-on utilisé ?

- Compiler le fichier `hello_xenomai.c` et installer l'application `hello_xenomai` dans le système de fichiers *root*.

- Se placer dans le répertoire `tst/jitter_xenomai/` :
```
host% cd tst/jitter_xenomai
```

- Analyser le fichier squelette `jitter_xenomai.c` donné en annexe et compléter le pour réaliser la mesure en continu du *jitter* sous forme d'une tâche Xenomai périodique, c'est-à-dire la mesure de la plus grande différence entre la mesure mesurée de l'incrémentation du *timer* et son incrémentation théorique sur une seconde (valeur mesurée sur une seconde moins 100 millions en dizaines de ns) . Quelle API Xenomai a-t-on utilisé ?

- Compiler le fichier `jitter_xenomai.c` et installer l'application `jitter_xenomai` dans le système de fichiers *root*.

- Regénérer le *RAM disk*.

21. EX 7 AMD : COMPILATION DU NOYAU LINUX XENOMAI

- Appliquer le patch Xenomai sur le noyau Linux. Que fait le *shell script* `go-ipipe` ? Quelle commande Linux utilise-t-on pour patcher un fichier ? :
```
host% cd xenomai
host% ./go-ipipe
```

- Compiler le noyau Linux Xenomai pour la carte cible ZedBoard :
```
host% cd linux-xenomai
host% ./go
```

- Installer le fichier du noyau Linux dans le répertoire de téléchargement d'*u-boot* `/tftpboot` :
```
host% ./goinstall
```

- Recharger si besoin le désign de référence `system.bit` dans le circuit FPGA de la carte cible ZedBoard :
```
host% cd ZedBoard
host% mbsdk2
[Xilinx EDK]$ ./load-design
```

- Depuis *u-boot* de la carte cible ZedBoard, exécuter la commande suivante :
```
U-Boot> run ramboot
```

- Observer les traces de boot du noyau Xenomai dans la fenêtre `minicom` :

```
Starting kernel ...

Uncompressing Linux... done, booting the kernel.
Booting Linux on physical CPU 0x0
Linux version 5.4.180-xilinx (kadionik@ipcchipik) (gcc version 11.3.0
(Buildroot 2021.11-4428-g6b6741b)) #3 SMP PREEMPT Wed Mar 15
15:01:00 CET 2023
CPU: ARMv7 Processor [413fc090] revision 0 (ARMv7), cr=18c5387d
CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing instruction cache
OF: fdt: Machine model: xlnx,zynq-7000
bootconsole [earlycon0] enabled
CPU: ARMv7 Processor [413fc090] revision 0 (ARMv7), cr=18c5387d
CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing instruction cache
OF: fdt: Machine model: xlnx,zynq-7000
bootconsole [earlycon0] enabled
earlycon: cdns0 at MMIO 0xe0001000 (options '115200n8')
bootconsole [cdns0] enabled
. . .
Kernel command line: console=ttyPS0,115200 root=/dev/ram
ramdisk_size=131072 rw
earlyprintk earlycon
. . .
I-pipe, 333.333 MHz clocksource, wrap in 12884 ms
clocksource: ipipe_tsc: mask: 0xffffffffffffffff max_cycles: 0x4ce07af025,
max_i
dle_ns: 440795209040 ns
. . .
timer #0 at (ptrval), irq=17
I-pipe, 333.333 MHz timer
Interrupt pipeline (release #8)
```

```
Console: colour dummy device 80x30
. . .
I-pipe, 333.333 MHz timer
. . .
[Xenomai] scheduling class idle registered.
[Xenomai] scheduling class rt registered.
I-pipe: head domain Xenomai registered.
[Xenomai] Cobalt v3.2.1
. . .
NET: Registered protocol family 17
Registering SWP/SWPB emulation handler
mmc0: new high speed SDHC card at address 0007
mmcblk0: mmc0:0007 SD04G 3.71 GiB
 mmcblk0: p1
hctosys: unable to open rtc device (rtc0)
ALSA device list:
  No soundcards found.
RAMDISK: gzip image found at block 0
EXT4-fs (ram0): mounted filesystem with ordered data mode. Opts: (null)
VFS: Mounted root (ext4 filesystem) on device 1:0.
devtmpfs: mounted
Freeing unused kernel memory: 1024K
Run /sbin/init as init process
    Hostname       : ZedBoard
    Kernel release : Linux 5.4.180-xilinx
    Kernel version : #3 SMP PREEMPT Wed Mar 15 15:01:00 CET 2023

 Mounting /proc             : [SUCCESS]
 Mounting /sys              : [SUCCESS]
 Mounting /dev              : [SUCCESS]
 Mounting /dev/pts          : [SUCCESS]
 Enabling hot-plug          : [SUCCESS]
 Populating /dev            : [SUCCESS]
 Mounting other filesystems : [SUCCESS]
 Starting telnetd           : [SUCCESS]
 Network configuration      : [SUCCESS]

System initialization complete.

Please press Enter to activate this console.

ZedBoard:/# uname -r
5.4.180-xilinx
ZedBoard:/#
```

22. EX 8 AMD : MESURE DE TEMPS DE LATENCE AVEC LE NOYAU LINUX XENOMAI

22.1. Outils standards

Nous allons mesurer des temps de latence sur le noyau Linux Xenomai dans le cas d'un noyau non stressé puis dans le cas d'un noyau stressé.

Pour stresser le noyau, on utilisera l'utilitaire `stress`.

- Dévalider le *throttling* :
  ```
  ZedBoard:# echo -1 > /proc/sys/kernel/sched_rt_runtime_us
  ```

- Dévalider l'anticipation sur la latence minimale de Xenomai :
  ```
  ZedBoard:# echo 0 > /proc/xenomai/latency
  ```

Noyau Xenomai non stressé. Outils standards :

- Lancer `cyclictest`. Noter le temps de latence maximum au bout de 5 minutes de tests :
  ```
  ZedBoard:# cyclictest -n -p 99 -i 5000
  ```

Noyau Xenomai stressé. Outils standards :

- Stresser le noyau avec `stress` :
  ```
  ZedBoard:# stress -c 50 -i 50 &
  ```

- Lancer `cyclictest`. Noter le temps de latence maximum au bout de 5 minutes de tests :
  ```
  ZedBoard:# cyclictest -n -p 99 -i 5000
  ```

Noyau Xenomai non stressé. Outils Xenomai :

On utilisera maintenant les outils Xenomai qui se trouvent dans le répertoire `/usr/xenomai/`.

- Lancer l'outil Xenomai `cyclictest`. Noter le temps de latence maximum au bout de 5 minutes de tests :
```
ZedBoard:# /usr/xenomai/demo/cyclictest -n -p 99 -i 5000
```

- On utilise maintenant l'outil Xenomai `latency` dans 3 modes différents. A quoi correspondent ces 3 modes ? Noter pour les 3 modes le temps de latence maximum au bout de 5 minutes de tests :
```
ZedBoard:# /usr/xenomai/bin/latency -t0 -p 5000
ZedBoard:# /usr/xenomai/bin/latency -t1 -p 5000
ZedBoard:# /usr/xenomai/bin/latency -t2 -p 5000
```

- Lancer `jitter_xenomai`. Noter le temps de latence maximum au bout de 5 minutes de tests :
```
ZedBoard:# jitter_xenomai
```

Noyau Xenomai stressé. Outils Xenomai :

- Stresser le noyau avec `stress` :
```
ZedBoard:# stress -c 50 -i 50 &
```

- Lancer l'outil Xenomai `cyclictest`. Noter le temps de latence maximum au bout de 5 minutes de tests :
```
ZedBoard:# /usr/xenomai/demo/cyclictest -n -p 99 -i 5000
```

- Lancer l'outil Xenomai `latency` dans les 3 modes. Noter pour les 3 modes le temps de latence maximum au bout de 5 minutes de tests :
```
ZedBoard:# /usr/xenomai/bin/latency -t0 -p 5000
ZedBoard:# /usr/xenomai/bin/latency -t1 -p 5000
ZedBoard:# /usr/xenomai/bin/latency -t2 -p 5000
```

- Lancer `jitter_xenomai`. Noter le temps de latence maximum au bout de 5 minutes de tests :
```
ZedBoard:# jitter_xenomai
```

22.2. Outils graphiques

Nous allons répéter les mesures avec des outils supplémentaires pour obtenir des graphiques. Nous allons uniquement exploiter `cyclictest`.

Nous aurons ainsi 2 types de graphiques :
- L'histogramme : ce graphique donne le nombre de fois que l'on obtient un temps de latence donné sur la durée de la mesure.

- La latence : ce graphique donne l'évolution du temps de latence au cours du temps. Si l'on a une mesure toutes les 1000 µs (1 ms), on en aura ainsi 1000 par seconde. On pourra alors visualiser l'évolution de la latence au cours du temps.

On ne produira ici que les graphiques dans le cas d'un noyau stressé.

Noyau Xenomai stressé. Outils standards :

- Stresser le noyau avec `stress` :
```
RPi3:# stress -c 50 -i 50 &
```

- Lancer `cyclictest` pour une mesure pour 5 minutes de tests. A quoi correspond la valeur 300000 ?
```
RPi3:# cyclictest -l 300000 -n -m -p 99 -i 1000 -v > ons.log
```

- Transférer le fichier `ons.log` (s pour standard) vers le PC hôte. Il faudra au préalable configurer l'interface réseau de la carte cible RPi (voir annexe 2) :
```
RPi3:# tftp -p -r ons.log @IP_host
```

- Recopier le fichier `ons.log` dans son répertoire de travail :
```
host% cd tst
host% cp /tftpboot/ons.log .
```

- Créer les graphiques histogramme et latence avec les *shells scripts* `gohist` et `golat` (sous */bin/*) créés par l'auteur des TP :
```
host% cd tst
host% cp tftpboot/ons.log .
host% gohist ons.log
host% golat ons.log
```

- Commenter les résultats obtenus.

Noyau Xenomai stressé. Outils Xenomai :

- Stresser le noyau avec `stress` :
```
RPi3:# stress -c 50 -i 50 &
```

- Lancer l'outil Xenomai `cyclictest` pour une mesure pour 5 minutes de tests :
```
RPi3:# /usr/xenomai/demo/cyclictest -l 300000 -n -m -p 99 -i
1000 -v > onx.log
```

- Transférer le fichier `onx.log` (x pour Xenomai) vers le PC hôte :
```
RPi3:# tftp -p -r onx.log @IP_host
```

- Recopier le fichier `onx.log` dans son répertoire de travail :
```
host% cd tst
host% cp /tftpboot/onx.log .
```

- Créer les graphiques histogramme et latence :

```
host% cd tst
host% cp tftpboot/onx.log .
host% gohist onx.log
host% golat onx.log
```

- Commenter les résultats obtenus. Les comparer aux résultats précédents.

23. CONCLUSION

On complètera le tableau suivant avec les mesures de temps de latence de la carte ZedBoard :

Outils Linux	**ZedBoard**	**ZedBoard**
Temps de latence en µs	**Linux standard non stressé**	**Linux standard stressé**
`cyclictest`		
`jitter`		

Outils Linux	**ZedBoard**	**ZedBoard**
Temps de latence en µs	**Linux Xenomai non stressé**	**Linux Xenomai stressé**
`cyclictest`		

Outils Xenomai	**ZedBoard**	**ZedBoard**
Temps de latence en µs	**Linux Xenomai non stressé**	**Linux Xenomai stressé**
`cyclictest Xenomai`		
`latency -t0`		
`latency -t1`		
`latency -t2`		
`jitter_xenomai`		

Que peut-on en conclure si l'on compare les résultats obtenus avec le noyau standard avec ceux obtenus avec le noyau Xenomai ?

Le matériel Libre *timer* 64 bits créé peut-il être utilisé comme outil de mesure de temps de latence ?

24. GRAND TP 3 : MISE EN ŒUVRE DE LA SYNTHESE DE HAUT NIVEAU HLS AVEC AMD ZYNQ

24.1. Introduction

L'usage d'un langage de description de matériel comme VHDL ou Verilog facilite le travail de synthèse apportant in fine un niveau d'abstraction significatif par rapport à l'approche traditionnelle sous forme de schémas électroniques. Ces langages de description de matériel permettent de travailler au niveau de la fonctionnalité RTL (*Register Transfer Logic*) comme les registres, additionneurs, comparateurs...

Ils sont aujourd'hui concurrencés par d'autres langages pour une synthèse de haut niveau HLS (*High Level Synthesis*) plus proche de l'algorithme et offrant un niveau d'abstraction bien supérieur pour mieux se focaliser sur la fonctionnalité et sur les flux d'échange de données à synthétiser.

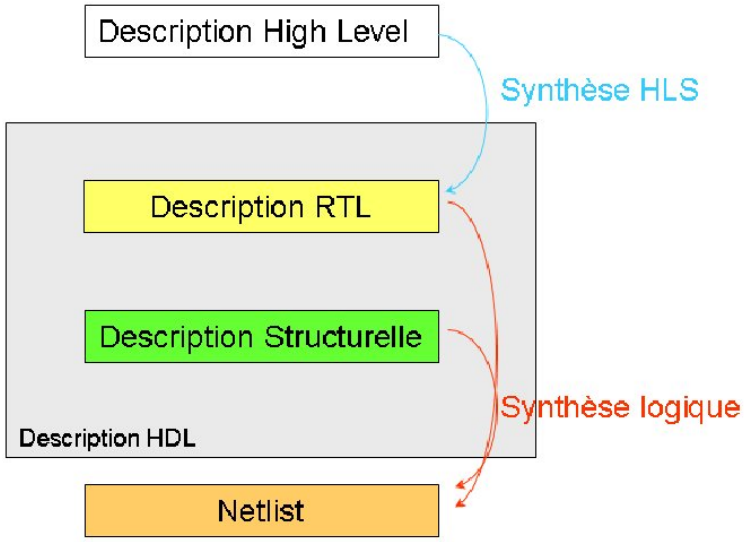
On peut citer comme langages courants pour la synthèse HLS :
- C et C++.
- SystemC. C'est une bibliothèque d'extension de C++.

Lequel choisir ? Si l'on veut focaliser sur l'aspect algorithmique, l'emploi de C ou C++ est approprié. Si l'on veut par contre avoir un contrôle plus strict sur les aspects temporels ou autres flux de données, SystemC est plus approprié.

La synthèse de haut niveau HLS (*High Level Synthesis*) pour la conception de circuits numériques permet de travailler à un niveau d'abstraction bien supérieur qu'il est possible de faire avec l'approche RTL classique en utilisant le langage VHDL par exemple. Cela permet au concepteur de s'affranchir des considérations d'implantation pour ne focaliser que sur sa fonctionnalité ou sur son algorithme.

On partira ainsi d'une description de l'algorithme en langage informatique C, C++ ou SystemC et le synthétiseur HLS produira en sortie une description RTL de l'algorithme en langage VHDL ou Verilog qui pourra être ensuite synthétisé avec un synthétiseur logique.

La figure suivante décrit le processus de synthèse HLS.



Synthèse HLS vs synthèse logique

Sur la figure précédente, la description structurelle est une description HDL de plus bas niveau qu'une description RTL basée sur les opérateurs logiques élémentaires (et, ou, bascules…). Elle est délaissée au profit de la description RTL. La *netlist* permet enfin de générer le fichier de programmation du circuit FPGA.

La synthèse HLS est basée sur 2 opérations importantes :

- L'ordonnancement (*scheduling*) : cette opération permet la translation du code C en opérations RTL. Cela se traduit par la création de la suite des opérations sur les données (*data path*) et du contrôle des opérations (*control path*) généralement à l'aide d'une machine d'états. Les décisions faites à ce niveau sont affectées par le choix de la période d'horloge, le choix de la cible FPGA mais aussi par les directives de synthèse HLS qu'applique le concepteur.
- La liaison (*binding*) : cette opération permet d'associer les opérations générées par le *scheduling* avec les ressources de la cible FPGA. Il y a aussi une rétroaction possible avec l'opération de *scheduling*.

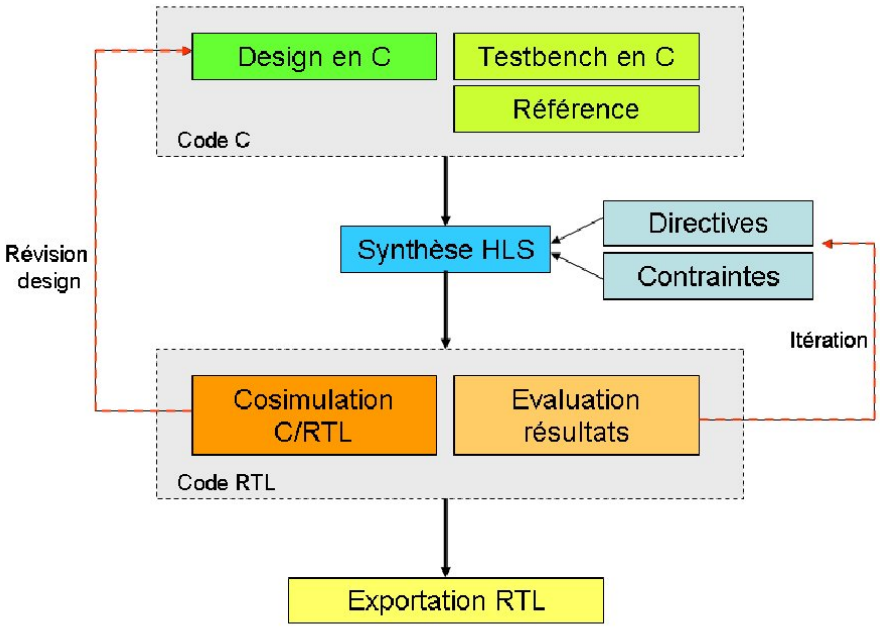
La synthèse HLS va ainsi traiter 2 aspects importants d'un *design* :

- L'interface du *design* : cela concerne les connexions externes au niveau *top-level*. Il y aura donc une synthèse HLS de l'interface.
- L'algorithme du *design* : il y aura donc une synthèse HLS de l'algorithme.

Cette synthèse HLS est grandement influencée par 2 paramètres :

- Les contraintes : contraintes sur le *timing* (valeur de la période d'horloge, incertitude sur cette période), éléments technologiques du circuit FPGA (blocs RAM, blocs DSP...)…
- Les directives : *pipelining* du *design*, parallélisme du *design*, choix d'un type d'interface pour le *design*...

L'outil Vitis HLS de AMD reprend ces grands principes. Le flot de conception avec l'outil Vitis HLS est décrit sur la figure suivante :



Flot de conception de Vitis HLS

Sur cette figure, on peut noter que l'on crée le *design* en utilisant le langage C (ou C++ ou SystemC). Un banc de test (*test bench*) écrit en langage C doit être développé et peut faire appel à des résultats de référence (*golden references*). Cela permet de simuler et de tester le code source C du *design*. Puis la synthèse HLS est réalisée conformément aux directives et aux contraintes imposées par le concepteur.

A l'issue de la synthèse HLS, on récupère une description RTL du *design*. On peut alors réaliser une cosimulation C/RTL en 3 étapes :

- Le *test bench* C est exécuté pour créer les stimuli qui serviront à tester le *design* RTL.
- Un *test bench* RTL est automatiquement généré pour appliquer uniquement les stimuli précédents au *design* RTL et pour récupérer ensuite les valeurs de sortie.

Les valeurs de sortie précédentes sont alors appliquées au *test bench* C pour vérifier les résultats obtenus avec ceux de référence.

On peut revenir au *design* C en cas de problèmes fonctionnels mais aussi réitérer la synthèse HDL en jouant sur les directives et sur les contraintes en fonction des objectifs fixés en termes de surface, de latence, de débit…

Enfin, le *design* RTL peut être exporté sous forme d'un bloc IP (I*ntellectual Property*) pouvant être ensuite importé dans un système SoPC développé avec l'outil AMD Vivado.

Les contraintes et les directives permettent d'influencer différents paramètres du *design* RTL dont on peut rappeler leur définition :

- Latence : temps en nombre de périodes d'horloge ou cycles (un *design* est toujours une logique synchrone sur front d'horloge) entre les valeurs appliquées à l'entrée du *design* et les valeurs produites en sortie.
- *Pipelining* : technique permettant d'optimiser le débit des données en entrée par l'ajout d'éléments mémoire (bascule D) à des endroits bien précis dans le *design*. Dans le meilleur des cas, le *design* accepte de nouvelles données à chaque cycle. Dans le cas d'un *pipelining*, AMD définit un intervalle d'initialisation II (*Initiation Interval*), qui est le nombre de cycles qui sépare l'acceptation de 2 données consécutives à l'entrée du *design*. Dans le meilleur des cas, II vaudra 1.
- Débit : c'est le débit des données acceptées en entrée par le *design*. Si l'on raisonne en temps, c'est nombre de cycles entre 2 données consécutives à l'entrée du *design*. S'il n'y a pas de *pipelining* utilisé, le débit en cycles est égal au temps de latence du *design*. Si l'on utilise la technique de *pipelining*, le débit en cycles est le II.

L'outil Vitis HLS permet de « jongler » avec ces paramètres grâce aux directives. On pourra en citer quelques unes et l'auteur renvoie le lecteur à la documentation AMD :

- Directives de réduction du temps de latence : LATENCY pour spécifier les valeurs min et max de la latence, UNROLL pour mettre à plat une boucle algorithmique (au lieu d'exécuter n fois le *design* interne d'une boucle, on instancie n fois le *design* interne qui est exécuté une fois (au détriment de la surface de circuit)), LOOP_FLATEN pour générer une seule boucle à partir de boucles imbriquées...
- Directive de *pipelining* : PIPELINE pour le *pipelining* d'une partie interne du *design*, DATAFLOW pour le *pipelining* du *top level* du *design*…
- Directives pour le choix des interfaces d'E/S.
- Directives pour le choix de l'implantation mémoire des paramètres d'entrée et de sortie du *design*.

Vitis HLS a enfin un ordre de priorité dans ses directives :

1. Atteindre les performances souhaitées : valeur de la période d'horloge, débit.
2. Minimiser la latence.
3. Minimiser la surface.

24.2. EX 9 AMD : développement et tests d'un algorithme par HLS

Nous allons créer avec Vitis HLS le périphérique matériel réalisant l'opération suivante :

`f(n) = 2n`^{2} `+ 3n + 5` avec `n` entier

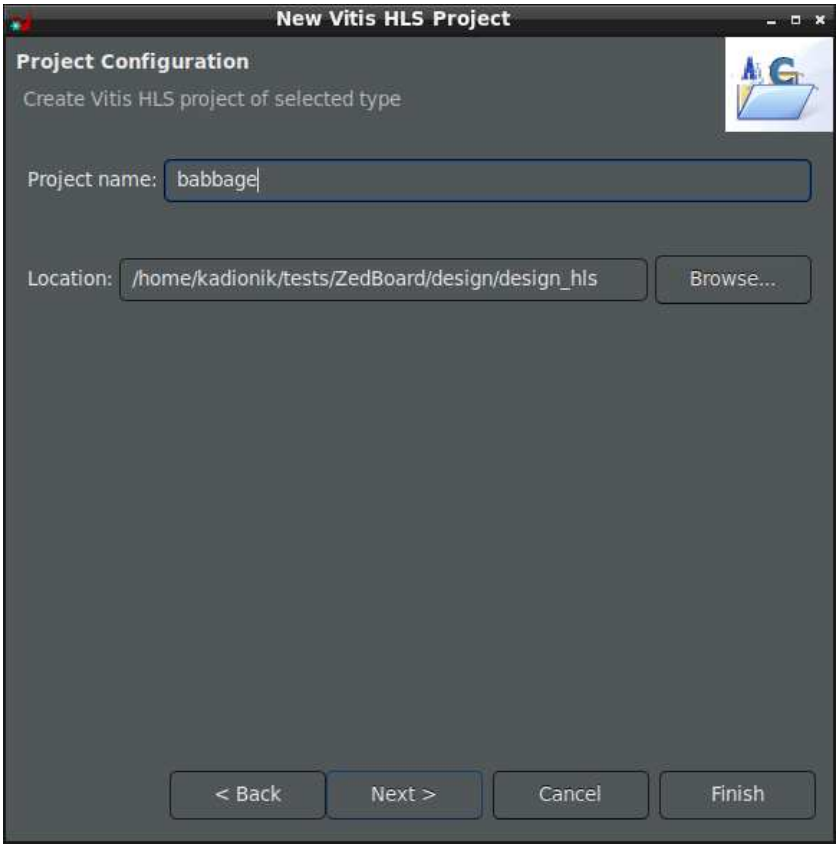
L'environnement Vitis HLS a déjà été vu dans le module EN325 « Flot de conception numérique avancée ». Nous passerons rapidement sur sa mise en œuvre…

- Se placer dans le répertoire `ZedBoard/` :
  ```
  host% cd ZedBoard
  ```

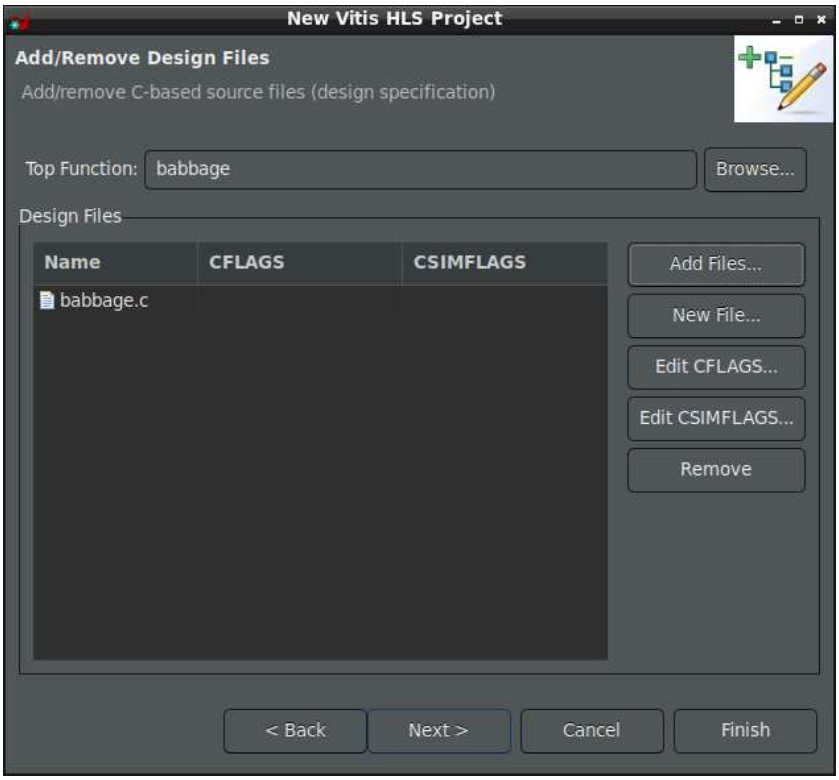
- Se placer ensuite dans le répertoire `design/design_hls/` et l'on se placera dans l'environnement AMD avec le *shell script* `mbsdk2` :
```
host% cd design/design_hls/
host% mbsdk2
```

- Lancer l'outil AMD Vitis HLS :
```
[Xilinx EDK]$ vitis_hls
```

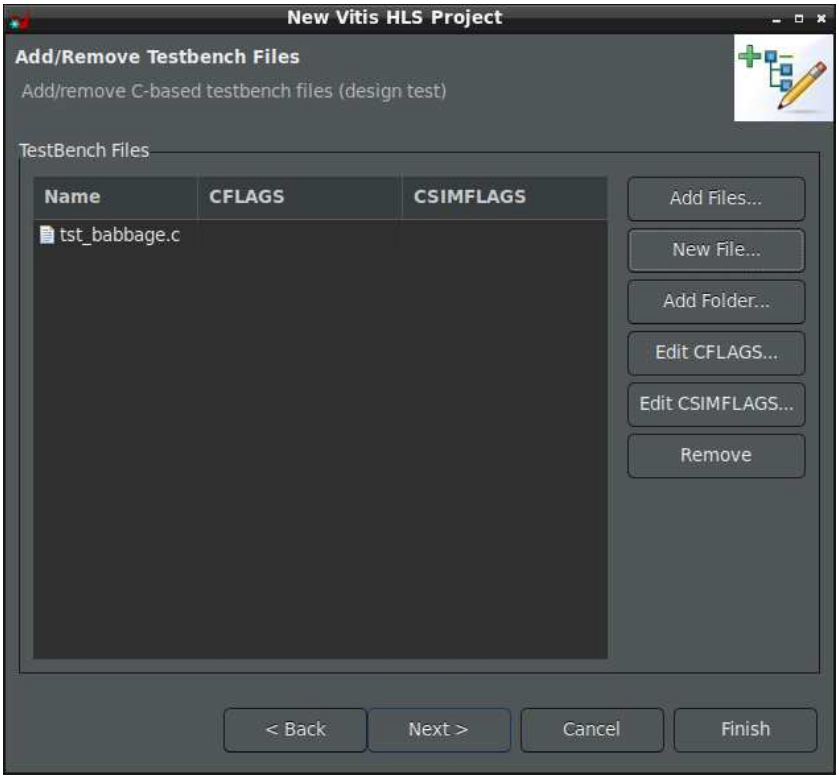
- Créer un projet appelé `babbage` en important le fichier source C `babbage.c`, le programme de test HLS `tst_babbage.c` et en choisissant comme cible la carte ZedBoard. Les figures suivantes illustrent les principales étapes :



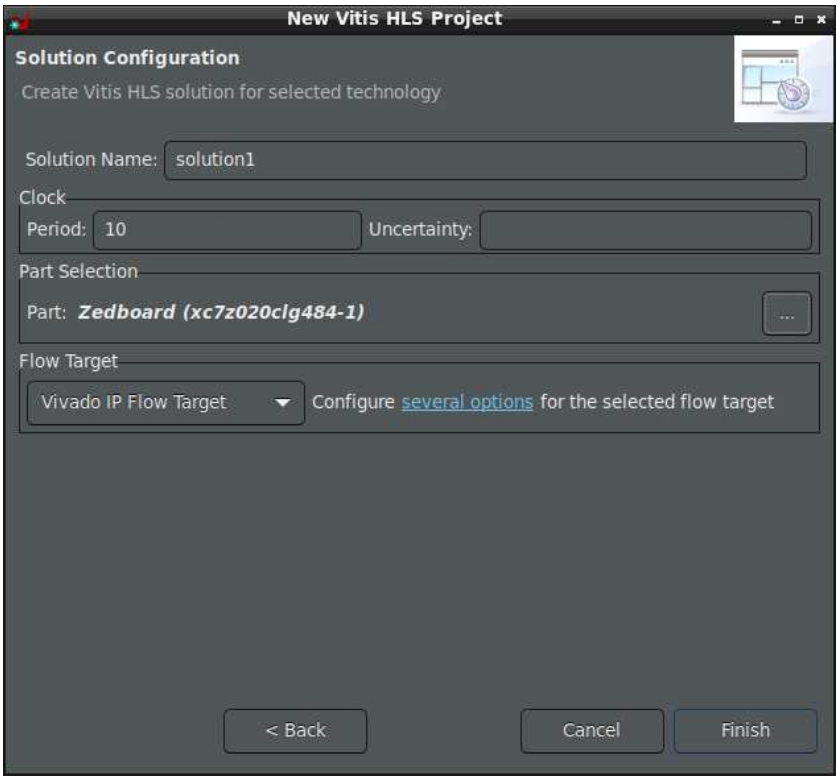
Création du projet HLS `babbage`



Inclusion du fichier source `babbage.c` de l'algorithme

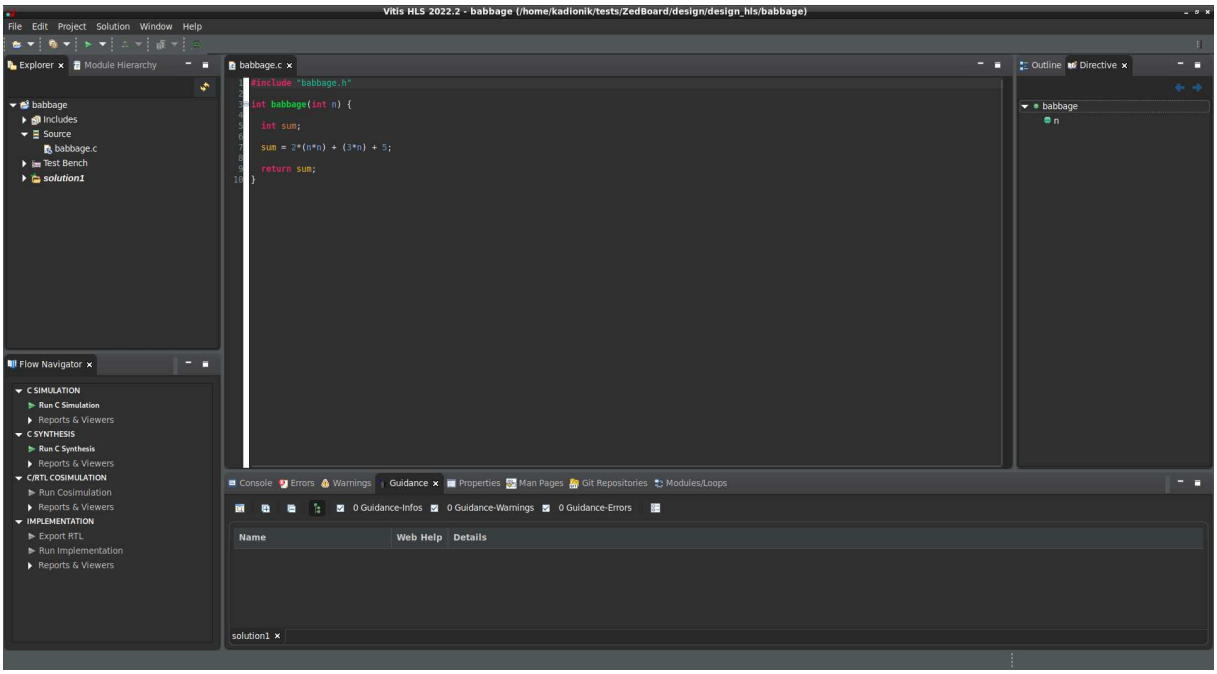


Inclusion du fichier source de test HLS `tst_babbage.c`



Choix de la cible ZedBoard

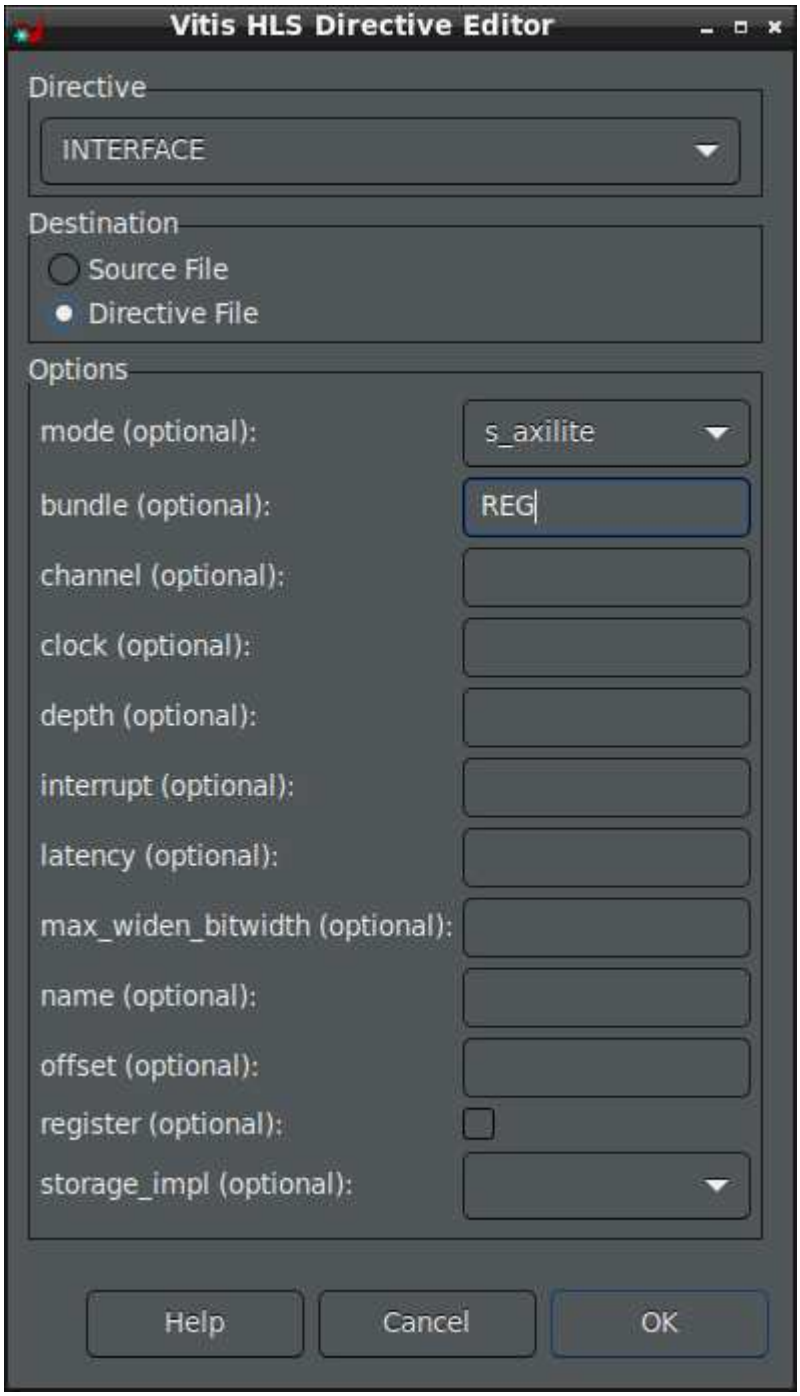
L'interface graphique de Vitis HLS est très simple et intuitive comme montré sur la figure suivante.



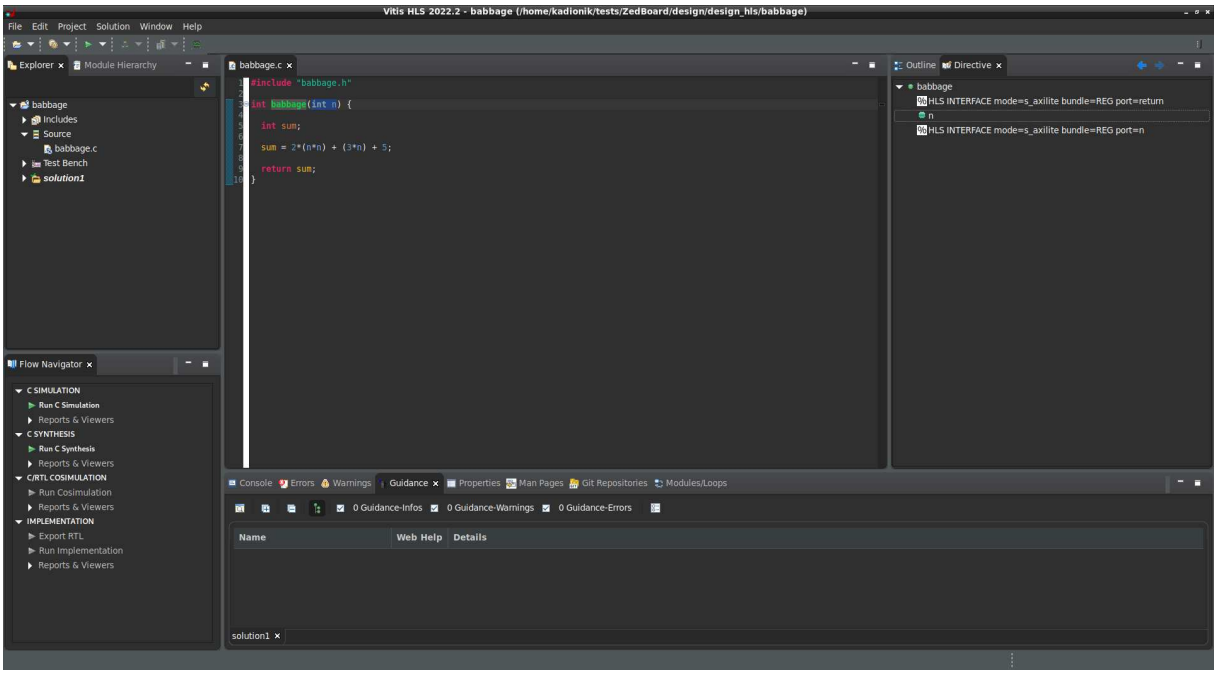
Interface graphique de Vitis HLS

- Modifier le fichier source C `babbage.c` pour implanter l'algorithme.

- Dans l'onglet *Directive*, définir (par clic droit de la souis) pour les paramètres d'E/S `babbage` et `n` accès en mode esclave (*s_axilite*) depuis le bus AXI du processeur du système SoPC. On initialisera la directive *bundle* à REG.



Directives HLS sur `babbage` et `n`



Directives HLS sur `babbage` et `n`

Le fichier `tst_babbage.c` est un fichier *test bench* pour la simulation C et la cosimulation C/RTL dont le code source est donné ci-après afin de comprendre la philisophie des tests.

```
#include <stdio.h>
#include "babbage.h"

#define TST_NBR 10
int main()
{
  int n;
  int hw_result[TST_NBR], sw_result[TST_NBR];
  int res;
  unsigned int err_cnt;
  int i;

  err_cnt = 0;

  // Software results
  n = 0;
  for (i=0; i<TST_NBR; i++) {
    res = 2*(n*n) + (3*n) + 5;
    sw_result[i] = res;
    n++;
  }

  // Call the DUT (hardware results)
  printf("Running DUT...");
  n = 0;
  for (i=0; i<TST_NBR; i++) {
    res = babbage(n);
    hw_result[i] = res;
    n++;
  }
  printf("done.\n");

  // Test the output against expected results
  printf("Testing DUT results...\n");
  for (i = 0; i < TST_NBR; i++) {
    printf("\tDUT   results:   test=%d,   DUT=%d,   expected=%d\n",   i,
hw_result[i],sw_result[i]);
    if (hw_result[i] != sw_result[i])
      err_cnt++;
  }

  if (err_cnt) {
    printf("----------Fail!-----------\n");
  }
  else {
    printf("----------Pass!-----------\n");

  }
}
```

Fichier source `tst_babbage.c`

Bien comprendre et expliquer le fonctionnement du fichier `tst_babbage.c`. Il faut Noter que Vitis HLS utilise le même fichier *test bench* pour la simulation C et la cosimulation C/RTL, ce qui simplifie la vie du concepteur.

- Lancer la simulation C par le menu *Project > Run C Simulation*. On obtient la figure suivante qui donne les résultats de la simulation C.

babbage.c babbage_csim.log

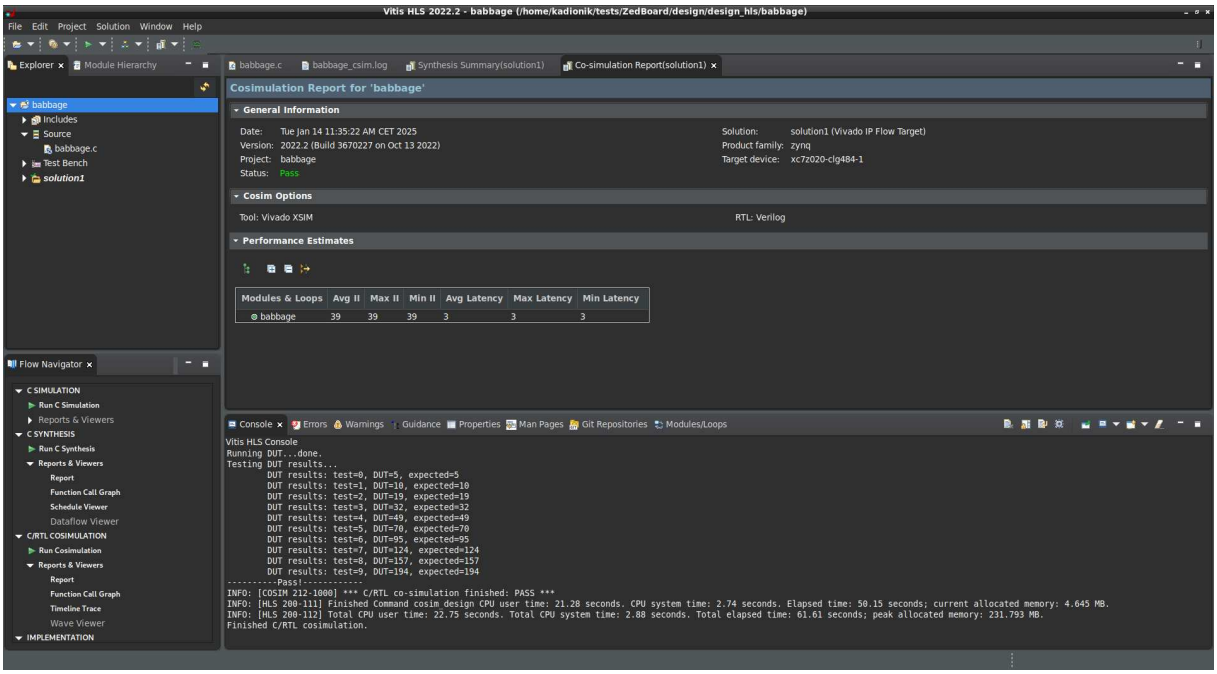
```
INFO: [SIM 2] ************** CSIM start **************
INFO: [SIM 4] CSIM will launch GCC as the compiler.
   Compiling(apcc) ../../../../tst_babbage.c in debug mode
INFO: [HLS 200-10] Running '/opt/Xilinx/Vitis_HLS/2022.2/bin/unwrapped/lnx64.o/apcc'
INFO: [HLS 200-10] For user 'kadionik' on host 'ipcchipik' (Linux_x86_64 version 6.12.8-200.fc41.x86_64) on Tue Jan 14 11:12
INFO: [HLS 200-10] On os "Fedora release 41 (Forty One)"
INFO: [HLS 200-10] In directory '/home/kadionik/tests/ZedBoard/design/design_hls/babbage/solution1/csim/build'
INFO: [APCC 202-3] Tmp directory is /tmp/apcc_db_kadionik/39591736849556048345
INFO: [APCC 202-1] APCC is done.
INFO: [HLS 200-112] Total CPU user time: 1.91 seconds. Total CPU system time: 0.13 seconds. Total elapsed time: 2.51 seconds
   Compiling(apcc) ../../../../babbage.c in debug mode
INFO: [HLS 200-10] Running '/opt/Xilinx/Vitis_HLS/2022.2/bin/unwrapped/lnx64.o/apcc'
INFO: [HLS 200-10] For user 'kadionik' on host 'ipcchipik' (Linux_x86_64 version 6.12.8-200.fc41.x86_64) on Tue Jan 14 11:12
INFO: [HLS 200-10] On os "Fedora release 41 (Forty One)"
INFO: [HLS 200-10] In directory '/home/kadionik/tests/ZedBoard/design/design_hls/babbage/solution1/csim/build'
INFO: [APCC 202-3] Tmp directory is /tmp/apcc_db_kadionik/40571736849559837210
INFO: [APCC 202-1] APCC is done.
INFO: [HLS 200-112] Total CPU user time: 1.79 seconds. Total CPU system time: 0.1 seconds. Total elapsed time: 1.94 seconds;
   Generating csim.exe
Running DUT...done.
Testing DUT results...
    DUT results: test=0, DUT=5, expected=5
    DUT results: test=1, DUT=10, expected=10
    DUT results: test=2, DUT=19, expected=19
    DUT results: test=3, DUT=32, expected=32
    DUT results: test=4, DUT=49, expected=49
    DUT results: test=5, DUT=70, expected=70
    DUT results: test=6, DUT=95, expected=95
    DUT results: test=7, DUT=124, expected=124
    DUT results: test=8, DUT=157, expected=157
    DUT results: test=9, DUT=194, expected=194
----------Pass!------------
INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] ************** CSIM finish **************
```

Simulation C

- Lancer la synthèse HLS en choisissant le menu *Solution > Run C Synthesis > Active Solution >*.

- Quels sont les résultats de synthèse (latence, période de l'horloge, éléments logiques utilisés...) ?

- Lancer la simulation C/RTL en choisissant le menu *Solution > C/RTL Cosimulation*. Expliquer comment marche cette cosimulation. On obtient la figure suivante qui donne les résultats de la cosimulation.



Cosimulation C/RTL

- Faire l'export du bloc IP généré par le menu *Solution >Export RTL.*

Le fichier du bloc IP généré se trouve dans le répertoire `design_hls/babbage/solution1/impl/ip/` et s'appelle ici `xilinx_com_hls_babbage_1_0.zip`.
Il pourra être importé par la suite dans le *design* SoPC.

24.3. EX 10 AMD : intégration d'un périphérique HLS

Nous allons réappliquer ce que l'on a vu dans le grand TP 2.

- Se placer dans le répertoire `ZedBoard/` :
  `host% cd ZedBoard`

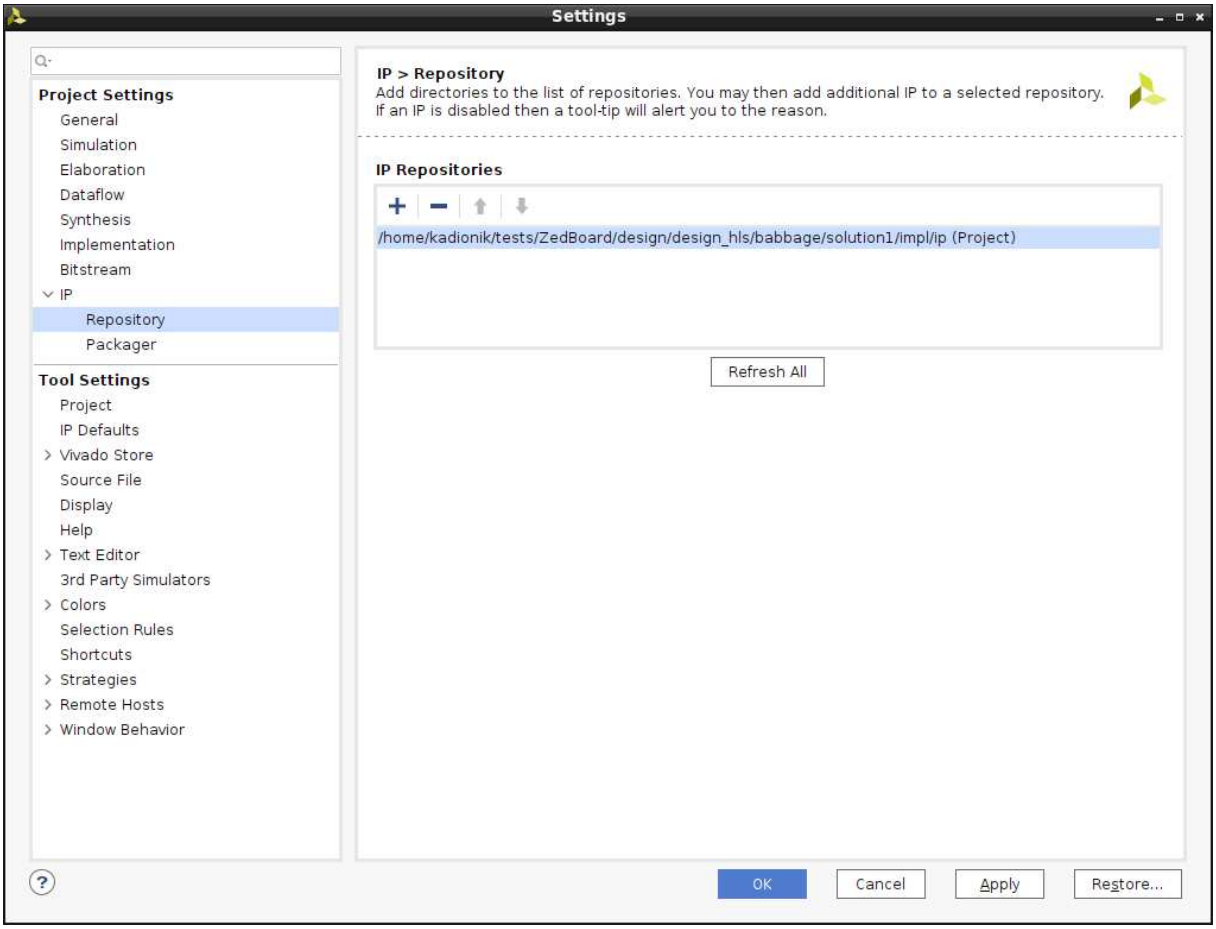
- Se placer ensuite dans le répertoire `design/design_hls/` et l'on se placera dans l'environnement AMD avec le *shell script* `mbsdk2` :
  `host% cd design/design_hls/`
  `host% mbsdk2`

- Lancer l'outil AMD Vivado de création de SoPC :
  `[Xilinx EDK]$ vivado`

- Ouvrir le projet Vivado `project_1.xpr` se trouvant dans le répertoire courant par :
  - *Open Project.*
  - Sélection de *project_1 > project_1.xpr.*

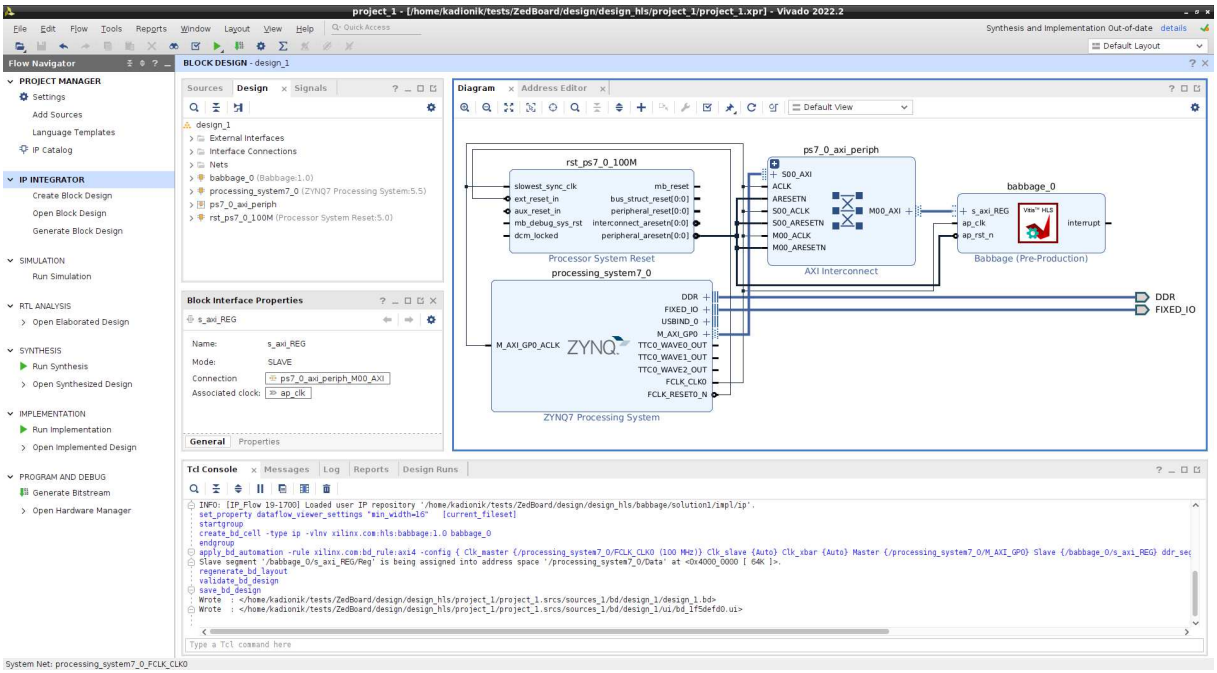
- Ouvrir le *Block Design* :
  - *Flow Navigator > IP Integrator > Open Block Design.*

- Rajouter l'accès au dépôt contenant le bloc IP `babbage` qui se trouve dans le répertoire `design_hls/babbage/solution1/impl/ip/` avec le menu *Flow Navigator* > *Project Manager* > *Settings* puis *Project Settings* > *IP* > *Repository*». On obtient la figure suivante.



Ajout du dépôt contenant le bloc IP `babbage`

- Ajouter au *design* le bloc IP `babbage`. On obtient alors le *design* final suivant comme montré sur la figure suivante.



Design final avec le bloc IP `babbage`

- Lancer enfin la synthèse pour générer le fichier `.bit` :
  - *Flow Navigator > Program and Debug > Generate Bitstream.*

- Sortir de Vivado. Recopier le nouveau fichier de programmation `system.bit` à la place de l'ancien :

```
host% cd ZedBoard
host% cp
design/design_hls/project_1/project_1.runs/impl_1/design_1_wrapper.bit system.bit
```

- Programmer le fichier `system.bit` dans le circuit FPGA de la carte cible ZedBoard. **Il faudra faire cette opération à chaque reset de la carte cible** :

```
host% cd ZedBoard
host% mbsdk2
[Xilinx EDK]$ ./load-design
```

24.4. Tests logiciels

- Se placer dans le répertoire `tst/babbage/` :

```
host% cd tst/babbage
```

- Analyser le fichier squelette `babbage.c` donné en annexe et compléter le pour tester le bloc IP `babbage` tel que cela a déjà été réalisé avec le testbench `tst_babbage.c`.

```
host% gedit tsttimer64.c
```

- Compiler le fichier `babbage.c` et tester le sur la carte ZedBoard.

25. CONCLUSION

Nous avons pu voir la mise en œuvre du SoPC avec les 2 principaux outils disponibles :

- Outil Quartus Prime d'Intel sur carte cible Terasic DE10-Standard.
- Outil Vivado et Vitis HLS d'AMD sur carte cible cible Digilent ZedBoard.

Nous avons pu réaliser 3 grands TP du matériel au logiciel embarqué mêlant à la fois les aspects matériels mais aussi les aspects logiciels :

- SoPC Intel : Quatus Prime, C embarqué, *bare metal*, µC/OS II.
- SoPC AMD : Vivado, Vitis HLS, C embarqué, Linux embarqué, Xenomai Cobalt.

Assemblées, toutes ces technologies forment un tout cohérent et serviront comme socle durant la carrière d'un ingénieur de l'embarqué…

26. REFERENCES

- Site d'Intel : https://www.intel.com/
- Site d'AMD : https://www.amd.com/fr/products/adaptive-socs-and-fpgas/fpga.html
- Carte DE10-Standard : https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=1081&PartNo=4
- DE10-Standard Computer System with Nios II : https://ftp.intel.com/Public/Pub/fpgaup/pub/Intel_Material/18.1/Computer_Systems/DE10-Standard/DE10-Standard_Computer_NiosII.pdf
- DE10-Standard User Manual : https://ftp.intel.com/Public/Pub/fpgaup/pub/Intel_Material/Boards/DE10-Standard/DE10_Standard_User_Manual.pdf
- Nios II Performance Benchmarks DS-1066 : https://cdrdv2-public.intel.com/666568/ds_nios2_perf-683629-666568.pdf
- The Zynq Book. L. Crockett and al. Editions Strathclyde Academic Media. Version numérique téléchargeable ici : http://www.zynqbook.com/
- Carte cible ZedBoard : http://zedboard.org/product/zedboard
- Les latences de Xenomai. C. Blaess : https://www.blaess.fr/christophe/2012/07/23/les-latences-de-xenomai/
- `mmap()` sur le site Intel : https://community.intel.com/t5/FPGA-Wiki/Accessing-hardware-registers-from-user-space-programs/ta-p/735151

27. ANNEXE 1 : FICHIER SOURCE TSTTIMER64.C

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <signal.h>
#include <sys/mman.h>

#include "xparameters.h"

#define SIZE 4096

volatile int *ptr;

int main(int argc, char **argv) {

  int fd;
  unsigned int fort1, fort2;
  unsigned int faible1, faible2;
  int delta;

  delta = 0;

  fd=open("/dev/mem",O_RDWR | O_SYNC);
  if(fd < 0) {
    printf("Failed to open /dev/mem\n");
    exit(-1);
  }
  printf("/dev/mem open OK\n");

  ptr    =    mmap(0,    SIZE,    PROT_READ|PROT_WRITE,    MAP_SHARED,    fd,
XPAR_MON_IP_0_S00_AXI_BASEADDR);

  if(ptr == (void *)-1) {
    close(fd);
    printf("mmap failed\n");
    exit(-1);
  }
  printf("mmap OK\n");

  printf("Test timer 64 bits. Delta value\r\n");
  printf("Delai = 1000 ms\r\n");

  *ptr = ???; // Reset

  while(1) {

    *ptr = ???; // Snapshot
    fort1 = ???;
    faible1 = ???;

    sleep(1);

    *ptr = ???; // Snapshot
    fort2 = ???;
    faible2 = ???;

    ...
```

```
  }

  munmap((void *)ptr, SIZE);
  close(fd);
  exit(0);
}
```

28. ANNEXE 2 : FICHIER SOURCE JITTER.C

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <signal.h>
#include <sys/mman.h>

#include "xparameters.h"

#define SIZE 4096

// 1 seconde correspond àl'incrémentation du compteur de 100 millions
#define COUNT_1S 100000000

// 1 microseconde correspond à l'incrémentation du compteur de 100 (10 ns)
#define COUNT_1U 100

volatile int *ptr;

int main() {
  int fd;
  unsigned int fort1, fort2;
  unsigned int faible1, faible2;
  unsigned long long value1, value2;
  int delta;
  unsigned int jitter;
  unsigned int jitter_max;

  jitter_max = 0;

  fd=open("/dev/mem",O_RDWR | O_SYNC);
  if(fd < 0) {
    printf("Failed to open /dev/mem\n");
    exit(-1);
  }
  printf("/dev/mem open OK\n");

  ptr    =    mmap(0,    SIZE,    PROT_READ|PROT_WRITE,    MAP_SHARED,    fd,
XPAR_MON_IP_0_S00_AXI_BASEADDR);

  if(ptr == (void *)-1) {
    close(fd);
    printf("mmap failed\n");
    exit(-1);
  }
  printf("mmap OK\n");

  *ptr = ???; // Reset

  while(1) {

    *ptr = ???; // Snapshot
    fort1 = ???;
    faible1 = ???;

    sleep(1);
```

```
    *ptr = ???; // Snapshot
    fort2 = ???;
    faible2 = ???;

    value1 = ((unsigned long long) fort1 << 32) | faible1;
    value2 = ((unsigned long long) fort2 << 32) | faible2;

    delta = value2-value1;
    jitter = abs(delta - COUNT_1S);

    ...
  }


  munmap((void *)ptr, SIZE);
  close(fd);
  exit(0);
}
```

29. ANNEXE 3 : FICHIER SOURCE HELLO_XENOMAI.C

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <signal.h>
#include <sys/mman.h>

#include <alchemy/task.h>

// Periode de 1 s
#define TIMESLEEP 1000000000

RT_TASK demo_task;
int end=0;

void catch_signal() {
  end=1;
}

void demo() {

  rt_printf("Starting Xenomai task...\n");

  // Configuration de la tache courante en mode periodique
  rt_task_set_periodic(NULL, TM_NOW, TIMESLEEP);

  while(end == 0) {

    // Attente de l'expiration de la periode
    rt_task_wait_period(NULL);

    rt_printf("Hello World from Xenomai!\n");
  }
}

int main() {
  char  str[32];

  // Execution de la fonction catch_signal Ctrl+C (SIGINT) et
  // Sur un kill -9 (SIGTERM) pour tuer la tache Xenomai
  signal(SIGTERM, catch_signal);
  signal(SIGINT, catch_signal);

  // Avoids memory swapping for this program
  mlockall(MCL_CURRENT|MCL_FUTURE);

  // Definition du nom de la tache Xenomai
  sprintf(str,"hello");

  // Creation et demarrage de la tache Xenomai
  rt_task_create(&demo_task, str, 0, 50, 0);
  rt_task_start(&demo_task, &demo, 0);

  // Attente appui sur clavier
  getchar();
```

```
  rt_task_delete(&demo_task);
  exit(0);
}
```

30. ANNEXE 4 : FICHIER SOURCE JITTER_XENOMAI.C

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <signal.h>
#include <sys/mman.h>

#include <alchemy/task.h>
#include "xparameters.h"

#define SIZE 4096

// 1 seconde correspond à l'incrémentation du compteur de 100 millions
#define COUNT_1S 100000000

// 1 microseconde correspond à l'incrémentation du compteur de 100 (10 ns)
#define COUNT_1U 100

// Periode de 1 s
#define TIMESLEEP 1000000000

// Priorite de la tache
#define PRIO 99

RT_TASK demo_task;
int end=0;

volatile int *ptr;

void catch_signal() {
  end=1;
}

void demo() {
  unsigned int fort1, fort2;
  unsigned int faible1, faible2;
  unsigned long long value1, value2;
  int delta;
  unsigned int jitter;
  unsigned int jitter_max;

  jitter_max = 0;

  rt_printf("Starting Xenomai task...\n");

  // Configuration de la tache courante en mode periodique
  rt_task_set_periodic(NULL, TM_NOW, TIMESLEEP);

  while(end == 0) {

    *ptr = ???; // Snapshot
    fort1 = ???;
    faible1 = ???;

    rt_task_wait_period(NULL);

    *ptr = ???; // Snapshot
    fort2 = ???;
```

```
    faible2 = ???;

    value1 = ((unsigned long long) fort1 << 32) | faible1;
    value2 = ((unsigned long long) fort2 << 32) | faible2;

    delta = value2-value1;
    jitter = abs(delta - COUNT_1S);

    ...
  }
}

int main() {
  char  str[32];
  int fd;

  signal(SIGTERM, catch_signal);
  signal(SIGINT, catch_signal);

  mlockall(MCL_CURRENT|MCL_FUTURE);

  fd=open("/dev/mem",O_RDWR | O_SYNC);
  if(fd < 0) {
    printf("Failed to open /dev/mem\n");
    exit(-1);
  }
  printf("/dev/mem open OK\n");

  ptr    =    mmap(0,    SIZE,    PROT_READ|PROT_WRITE,    MAP_SHARED,    fd,
XPAR_MON_IP_0_S00_AXI_BASEADDR);

  if(ptr == (void *)-1) {
    close(fd);
    rt_printf("mmap failed\n");
    exit(-1);
  }
  printf("mmap OK\n");

  *ptr = 0x2; // Reset

  sprintf(str,"jitter");

  rt_task_create(&demo_task, str, 0, PRIO, 0);
  rt_task_start(&demo_task, &demo, 0);

  // Attente appui sur clavier
  getchar();

  rt_task_delete(&demo_task);
  munmap((void *)ptr, SIZE);
  close(fd);
  exit(0);
}
```

31. ANNEXE 5 : FICHIER SOURCE BABBAGE.C

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <signal.h>
#include <sys/mman.h>

#include "xparameters.h"
#include "xbabbage_hw.h"

#define TST_NBR 10
#define AP_START 0x01
#define AP_STOP 0x00
#define AP_DONE 0x02
#define SIZE 4096

volatile int *ptr;

int start() {
  *(ptr + (XBABBAGE_REG_ADDR_AP_CTRL/4)) = AP_START; // Offset registre en
octets à traduire en 32 bits
}

int stop() {
  *(ptr + (XBABBAGE_REG_ADDR_AP_CTRL/4)) = AP_STOP;
}

int wait() {
  while((*(ptr + (XBABBAGE_REG_ADDR_AP_CTRL/4)) & AP_DONE) != AP_DONE) {
    ;
  }
}

int main(int argc, char **argv) {

  int fd;
  int n;
  int hw_result[TST_NBR], sw_result[TST_NBR];
  int res;
  unsigned int err_cnt;
  int i;

  fd=open("/dev/mem",O_RDWR | O_SYNC);
  if(fd < 0) {
    printf("Failed to open /dev/mem\n");
    exit(-1);
  }
  printf("/dev/mem open OK\n");

  ptr = mmap(0, SIZE, PROT_READ|PROT_WRITE, MAP_SHARED, fd,
XPAR_BABBAGE_0_S_AXI_REG_BASEADDR);

  if(ptr == (void *)-1) {
    close(fd);
    printf("mmap failed\n");
    exit(-1);
  }
```

```
  printf("mmap OK\n");

  stop();

  err_cnt = 0;

  // Software results
  n = 0;
  for (i=0; i<TST_NBR; i++) {
    . . .
  }

  // Call the DUT (hardware results)
  printf("Running DUT...");
  n = 0;
  for (i=0; i<TST_NBR; i++) {
    . . .
  }
  printf("done.\n");

  // Test the output against expected results
  printf("Testing DUT results...\n");
  for (i = 0; i < TST_NBR; i++) {
    . . .
  }

  if (err_cnt) {
    printf("----------Fail!-----------\n");
  }
  else {
    printf("----------Pass!-----------\n");

  }
  munmap((void *)ptr, SIZE);
  close(fd);
  exit(0);
}
```

32. ANNEXE 6 : CONFIGURATION RESEAU HOTES ET CIBLES

	POSTE PC01	
	NOM	**ADRESSE IP**
HOTE	citron07	192.168.4.7
CIBLE	ml01 zb01	192.168.4.101

	POSTE PC02	
	NOM	**ADRESSE IP**
HOTE	citron08	192.168.4.8
CIBLE	ml02 zb02	192.168.4.102

	POSTE PC03	
	NOM	**ADRESSE IP**
HOTE	citron09	192.168.4.9
CIBLE	ml03 zb03	192.168.4.103

	POSTE PC04	
	NOM	**ADRESSE IP**
HOTE	citron10	192.168.4.10
CIBLE	ml04 zb04	192.168.4.104

	POSTE PC05	
	NOM	**ADRESSE IP**
HOTE	citron11	192.168.4.11
CIBLE	ml05 zb05	192.168.4.105

	POSTE PC06	
	NOM	**ADRESSE IP**
HOTE	citron12	192.168.4.12
CIBLE	ml06 zb06	192.168.4.106

Masque de sous réseau : **255.255.255.0**

Exemple : configuration réseau de la carte cible `ml01` :

```
target# ifconfig eth0 192.168.4.101
```