Projet FPGA

Notions élémentaires, simulation et synthèse

IESE5 – Polytech Grenoble

Objectif

Ce projet a pour but d'introduire les notions de base pour l'implémentation d'un système numérique sur FPGA.

Outils

— **nedit/gedit/vim** : éditeur de texte utilisé pour la création et édition des descriptions VHDL des systèmes numériques.

— **GHDL** : simulateur open-source pour le langage VHDL. Il permet de compiler et d'exécuter votre code directement dans votre PC (http://ghdl.free.fr/).

— **GTKWave** : visualiseur de formes d'ondes basé sur GTK (http://gtkwave.sourceforge.net/).

— **Vivado Design Suite** : suite logicielle Xilinx pour la synthèse et l'analyse de modèles HDL. Vivado permet le développement de systèmes sur puce et la synthèse de haut niveau. Il existe une version gratuite (Vivado WebPACK Edition) qui offre aux concepteurs une version limitée de l'environnement de conception (https://www.xilinx.com/products/design-tools/vivado.html).

1 Environnement de travail

Vous trouverez dans votre compte un répertoire `projet_fpga` contenant un ensemble de fichiers sources de départ. Ce répertoire sera considéré comme le répertoire de travail racine pour l'ensemble des exercices proposés dans ce document. Vous trouverez initialement les informations présentées ci-dessous.

— **Fichiers `.vhd`** : fichiers sources vhdl contenant les modèles de quelques entités et architectures requises pour le projet. Certains de ces fichiers sont incomplets et d'autres sont prêts pour l'instanciation (à vous de les identifier).

— **Sous-répertoires de test** : répertoires des fichiers de test associés à un composant. Chaque répertoire de test est identifié avec le nom de l'entité correspondante suivi du suffixe `_tb`. Ils contiennent :

  — L'entité et l'architecture des testbench (`.vhd`)

  — Le fichier `Makefile` permettant de compiler et simuler le composant sous test.

— **Sous-répertoire `_vivado`** : répertoire d'exemple contenant la structure et les fichiers de contraintes requis pour la création d'un projet dans Vivado. Les fichiers contenus dans ces répertoires doivent aussi être complétés.

  — `prj` : sous-répertoire permettant de stocker les fichiers générés automatiquement par l'outil de synthèse.

  — `src` : sous-répertoire contenant le modèle de plus haut niveau à implémenter sur FPGA (`.vhd`).

  — `xdc` : sous-répertoire contenant les fichiers de contraintes physiques et de temps associés au design (`.xdc`).

— **Sous-répertoire `doc`** : répertoire contenant la documentation de la carte FPGA à utiliser dans ce projet et un exemple d'un fichier de contraintes préconfiguré pour la Zybo (Carte de développement Zynq-7000).

2 Prise en main des outils : Conception et simulation d'un registre d'un bit

Dans cette première partie vous devez décrire l'entité `reg` et son architecture dans le fichier `reg.vhd` (situé dans la racine de votre répertoire de travail). Vous devez respecter l'interface proposée dans la Figure 1 et aussi définir un banc de test afin de valider votre composant.

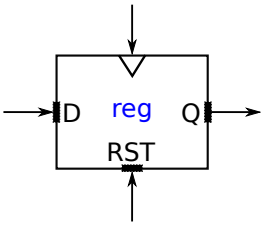

Figure 1 – Registre d'un bit

Question 2.1. Modélisez un registre `reg` d'un bit en VHDL ayant les entrées/sorties suivantes :

— Une horloge `CLK` et un port asynchrone `RST` représentant le reset (actif haut) de type `std_logic`.

— Une entrée `D` de type `std_logic` contenant la valeur à stocker.

— Une sortie `Q` de type `std_logic` qui sera initialisée à zéro pendant la phase de reset.

Le comportement de votre registre sera implémenté à l'intérieur d'un processus sensible aux ports `CLK` et `RST`.

Question 2.2. Construisez en VHDL un modèle de test `reg_tb` pour simuler votre registre. L'entité de test doit respecter l'interface proposée dans la Figure 2. Cette entité sera définie dans le fichier `reg_tb/reg_tb.vhd`.

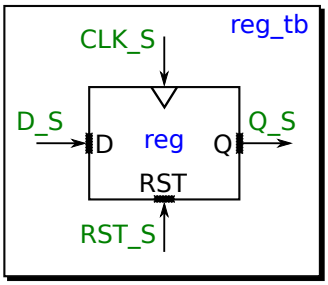

Figure 2 – Entité de test pour un registre d'un bit

Question 2.3. Compilez, élaborez et simulez votre composant à l'aide du Makefile défini dans le répertoire `reg_tb`. Pour cela vous devez vous déplacer dans le répertoire et utiliser les commandes suivantes :

```
$ make
$ make run
$ make view
```

Attention : Observez la structure du `Makefile` et identifiez les différentes étapes, ainsi que les bibliothèques où sont analysés les composants. Dans le cas d'un modèle de type structural, vous devrez analyser d'abord tous les composants internes avant d'analyser le composant de plus haut niveau.

Après l'exécution de la dernière commande, vous obtiendrez une fenêtre similaire à celle de la Figure 3. À gauche vous pouvez sélectionner les signaux à ajouter dans le chronogramme de droite. Comme vous l'avez sûrement identifié, le temps de simulation est spécifié directement dans le fichier `Makefile`.

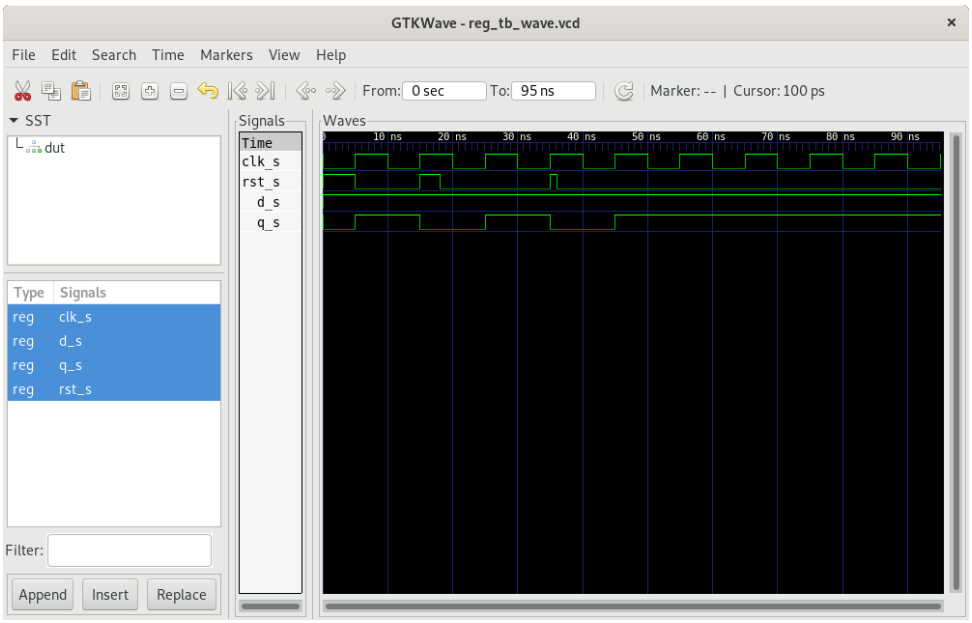

Figure 3 – Résultat de la simulation du registre d'un bit

Question 2.4. Confirmez-vous que le résultat sur les signaux de sortie est conforme à ce qui était attendu ? Pourquoi ? Expliquez en détail les résultats sur votre rapport.

Attention : Après avoir simulé et vérifié le comportement de votre composant, nettoyez votre répertoire de test à l'aide de la commande `make clean`.

3 Mise en pratique : Conception et simulation d'un synchroniseur de reset

On se propose dans cette partie de faire un synchroniseur de reset que nous pourrons utiliser plus tard dans notre projet.

Un synchroniseur de reset est un composant indispensable pour éviter la métastabilité dans les systèmes fonctionnant avec des signaux de reset asynchrones. Dans ce projet, tous les composants à utiliser doivent être implémentés avec des signaux de reset asynchrones (actifs haut). Cela signifie que, par exemple, lorsque le signal `RST = 1` tous les registres et machines à états iront vers la valeur initiale. Ce choix a été fait avec un objectif pédagogique.

Vous vous demanderez peut être quand est produit un état de métastabilité ? Dans un système avec reset asynchrone, comme son nom l'indique, le signal de reset `RST` peut être activé ou désactivé indépendamment du signal d'horloge `CLK`. Dans les cas où le signal `RST` est désactivé au même temps qu'un front montant d'horloge arrive, la métastabilite peut se produire, c'est-à-dire un état dont la valeur des registres impliqués sera inconnu. Observez ce comportement sur la Figure 4.

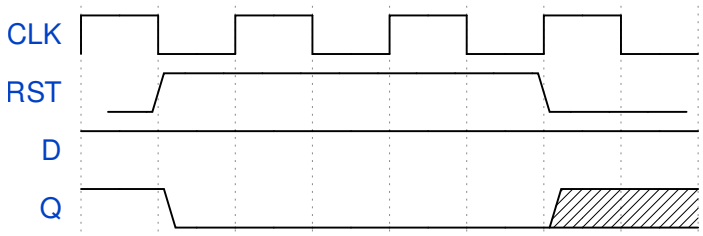

Figure 4 – Métastabilité dans un système avec reset asynchrone actif haut (Figure 1)

La solution pour éviter ce comportement, est de synchroniser à l'horloge le passage du signal de reset de l'état actif à l'état inactif. Cette solution peut être implémentée à l'aide du composant proposé dans la Figure 5.

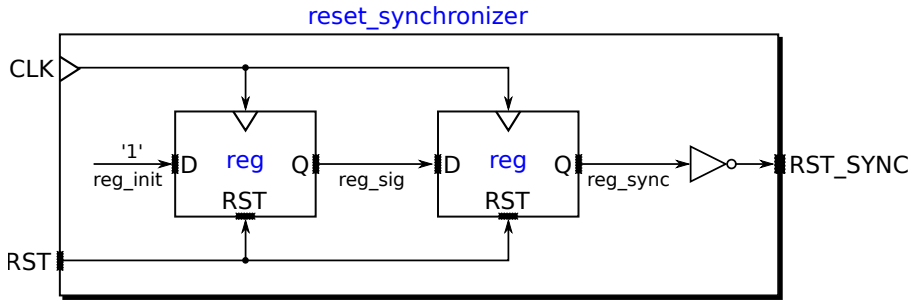

Figure 5 – Synchroniseur de reset (actif haut)

Question 3.1. Modélisez un synchroniseur de reset `reset_synchronizer` en VHDL en respectant l'interface et l'architecture définies dans la Figure 5. Pour cela vous réutiliserez le registre d'un bit défini et validé dans la section précédente. Expliquez en détail le comportement du système quand le signal `RST = 1` et aussi le cas contraire. Rappelez vous, toutes les réponses doivent être justifiées dans votre rapport.

Pour expliquer le comportement du système, vous pouvez vous inspirer du chronogramme de la Figure 6, dont le changement du `RST_SYNC` est produit à cause du front montant d'horloge.

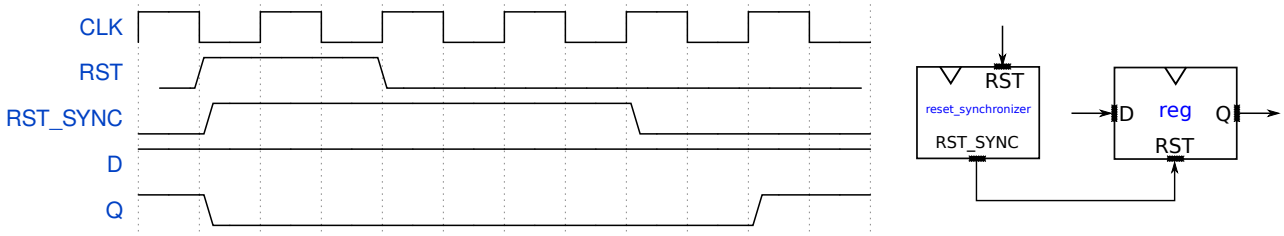

Figure 6 – Synchronisation à l'horloge du reset

Question 3.2. Construisez en VHDL une entité de test permettant de valider le comportement de votre composant en simulation. Votre banc de test doit se comporter comme indiqué sur la Figure 7.

Question 3.3. Confirmez-vous que le résultat sur les signaux de sortie est conforme à ce qui était attendu ? Pourquoi ? Expliquez en détail les résultats sur votre rapport.

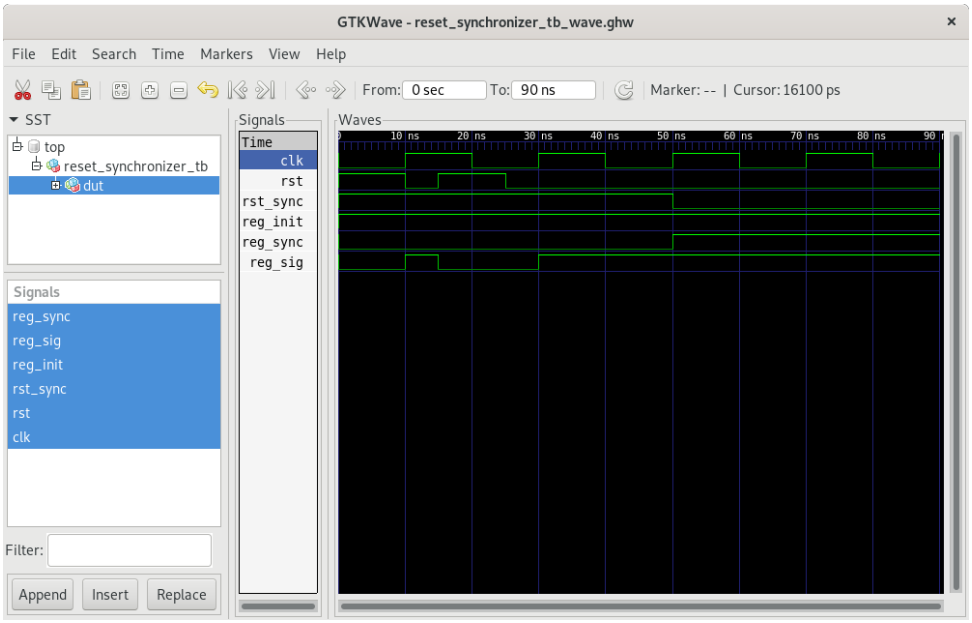

Figure 7 – Résultat de la simulation du synchroniseur de reset

4 Prise en main des outils : Prototypage FPGA

Dans cette partie on introduit l'utilisation de l'outil Vivado. Nous allons effectuer la synthèse et l'implémentation du modèle du synchroniseur de reset sur une carte FPGA Zybo. Afin de pouvoir effectuer le prototypage, vous devez d'abord créer une entité de plus haut niveau, qu'on appellera `reset_synchronizer_top`, respectant l'interface définie dans la Figure 8.

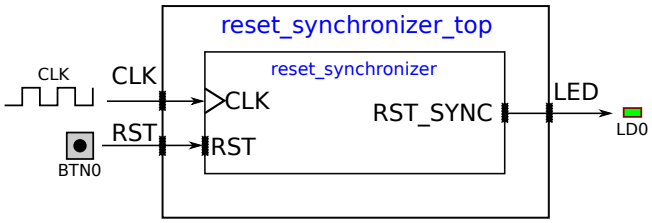

Figure 8 – Entité de plus haut niveau requise pour le prototypage sur carte FPGA

Question 4.1. Créez l'entité de la Figure 8 dans le répertoire `reset_synchronizer_vivado/src/`. Notez qu'il s'agit simplement de l'instanciation du composant `reset_synchronizer` (modélisé dans la section précédente) dans une entité de plus haut niveau.

Attention : La structure du répertoire utilisé pour la synthèse est la suivante (cette structure sera utilisé à chaque fois qu'on souhaite réaliser une nouvelle synthèse) :

```
reset_synchronizer_vivado/
|-- prj/
| \_ ... vide au debut (utilise pour la creation du projet Vivado)
|-- src/
| \_ reset_synchronizer_top.vhd (a completer)
|-- xdc/
   |_ physical.xdc (a completer -> contraintes physiques du design)
   |_ timing.xdc (a completer -> contraintes de temps du design)
   \_ generated.xdc (fichier utilise par Vivado - a ne pas modifier)
```

La deuxième étape avant de commencer la synthèse est de créer les fichiers de **contraintes physiques** et de **contraintes de temps** associés au design. Pour cela vous devez utiliser comme référence le fichier `.xdc` disponible dans le répertoire `doc`. Ce fichier est fourni par le fabricant de la carte.

Question 4.2. Modifiez le fichier `physical.xdc` afin d'interconnecter les PINS du FPGA avec les entrées/-sorties du modèle.

- Interconnectez le signal d'horloge en utilisant la commande `set_property`. Les lignes fournies dans le fichier `.xdc` de référence permettent d'utiliser l'horloge du FPGA à 125MHz (PIN `L16` ou `K17` selon le modèle du FPGA utilisé : Zybo ou Zybo Z7) et de l'interconnecter avec le port `CLK` du modèle.
- Interconnectez les boutons poussoirs et les leds :
  - Utilisez le bouton `BTN0(R18)` ou `BTN0(K18)` (selon le FPGA) pour le reset `RST`
  - Utilisez la led `LD0(M14)` pour afficher la sortie du modèle `LED`

Attention : Vous pouvez vérifier tous les PINS dans la documentation disponible dans votre répertoire de travail.

Question 4.3. Modifiez le fichier `timing.xdc` afin de créer l'horloge et d'indiquer au synthétiseur quels chemins ne doivent pas être vérifiés par l'analyseur de timing. C'est le cas des boutons poussoirs et des leds car ces signaux sont asynchrones (ne sont liés à aucun signal d'horloge).

- Créez l'horloge externe 125MHz (commande `create_clock`)
- Indiquez les chemins à ne pas vérifier (commande `set_false_path`) (dans votre cas `RST` et `LED`)

4.1 Création d'un projet dans Vivado

- Avant de commencer ouvrez un terminal, exécutez le script de configuration, accédez à votre répertoire de travail `prj` et lancez Vivado :

```
$ source settings_vivado_2017_1.sh
$ cd projet_fpga/reset_synchronizer_vivado/prj
$ vivado &
```

- Accédez à `File -> Project -> New ...` et suivez les instructions indiquées sur chaque fenêtre. Voici un récapitulatif des étapes :
  - Création d'un nouveau projet
  - Sélection du nom de votre projet Vivado (par exemple, `projet_reset_synchronizer`) et sélection du dossier de travail (`prj`). Attention à ne pas créer de sous-répertoire pour le projet
  - Sélection du type de projet, RTL dans votre cas.
  - Ajout des fichiers sources (Figure 9). Dans cette étape vous devez ajouter tous les fichiers VHDL nécessaires pour le fonctionnement de votre design. Attention à ne pas copier les sources dans le projet et à sélectionner le langage VHDL comme langage de simulation.
  - Ajout des fichiers de contraintes (Figure 10). Dans cette étape vous devez ajouter tous les fichiers `xdc` définis précédemment.
  - Sélection du FPGA à utiliser (Figure 11) : Zybo `xc7z010clg400-1` ou `xc7z020-1clg400c` (modèle à voir sur votre carte).
  - Vérification du récapitulatif du projet.

— Une fois le projet crée, vous pouvez vérifier la hiérarchie de votre projet et tout à gauche identifier les étapes à suivre (Figure 12).

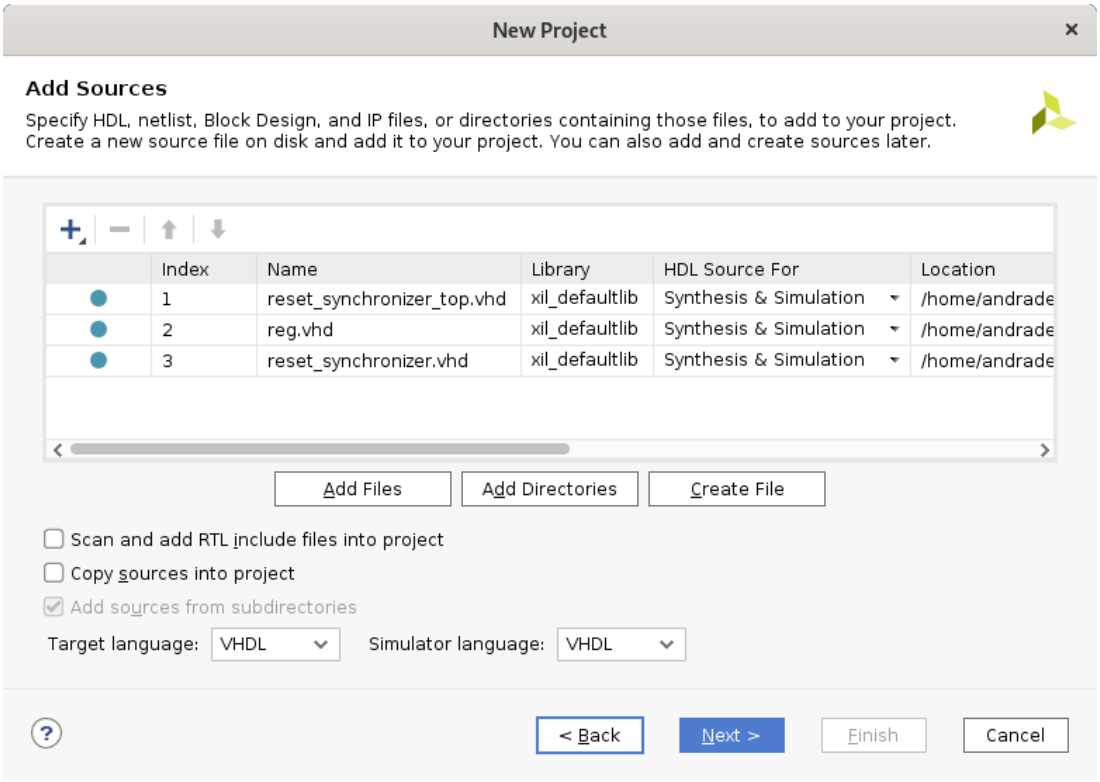

Figure 9 – Ajout des sources dans le projet Vivado

4.2 Analyse RTL

Une fois le projet crée vous pouvez accéder à `RTL Analysis -> Open Elaborated Design -> Schematic` et vérifiez votre design (Figure 13).

Question 4.4. Obtenez-vous le résultat attendu ? Explorez votre design en utilisant l'icone +. Commentez vos résultats.

4.3 Synthèse

— Allez dans `Synthesis -> Run synthesis` (choisissez les options par défaut)

— Attendez les résultats et ouvrez le modèle synthétisé. Utilisez le zoom pour retrouver les pins qui correspondent aux entrées/sorties de votre design. Vous pouvez aussi accéder à ce modèle dans le menu `Window -> Device`

— Observez les résultats de la synthèse dans l'onglet `Projet Summary`. Attention aux warnings (tous ne peuvent pas être résolus).

Question 4.5. Retrouvez-vous les entrées/sorties ? (Figure 14). Commentez vos résultats.

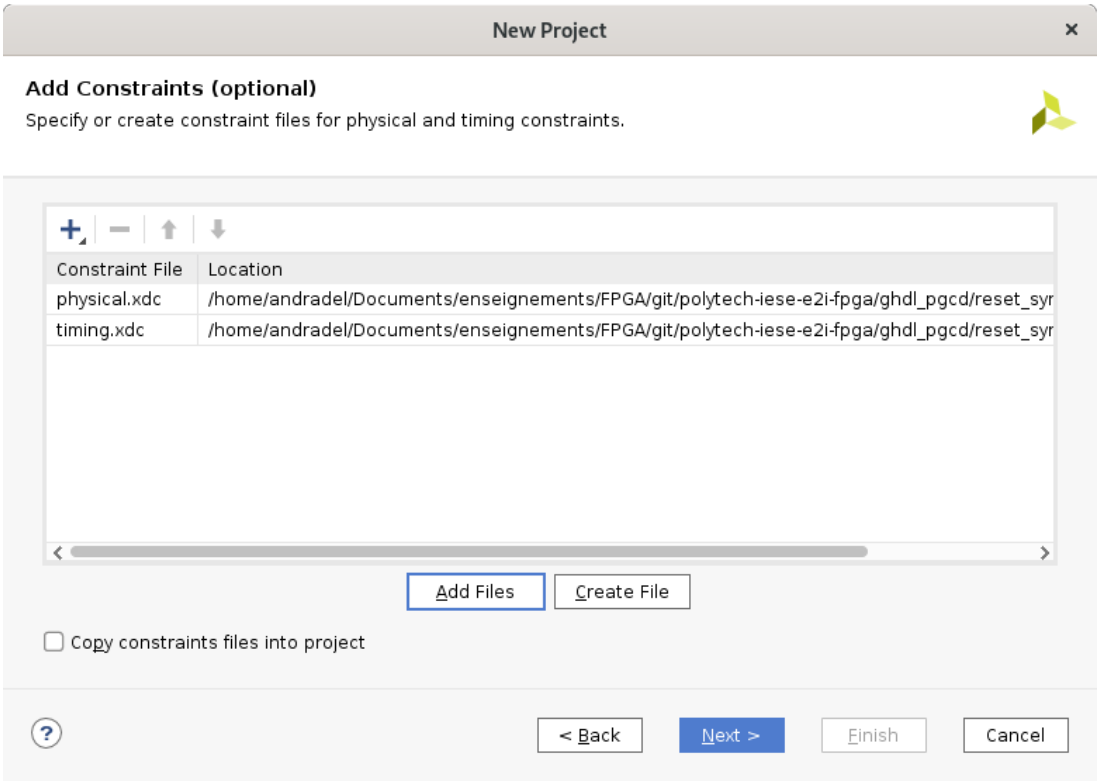

Figure 10 – Ajout des contraintes dans le projet Vivado

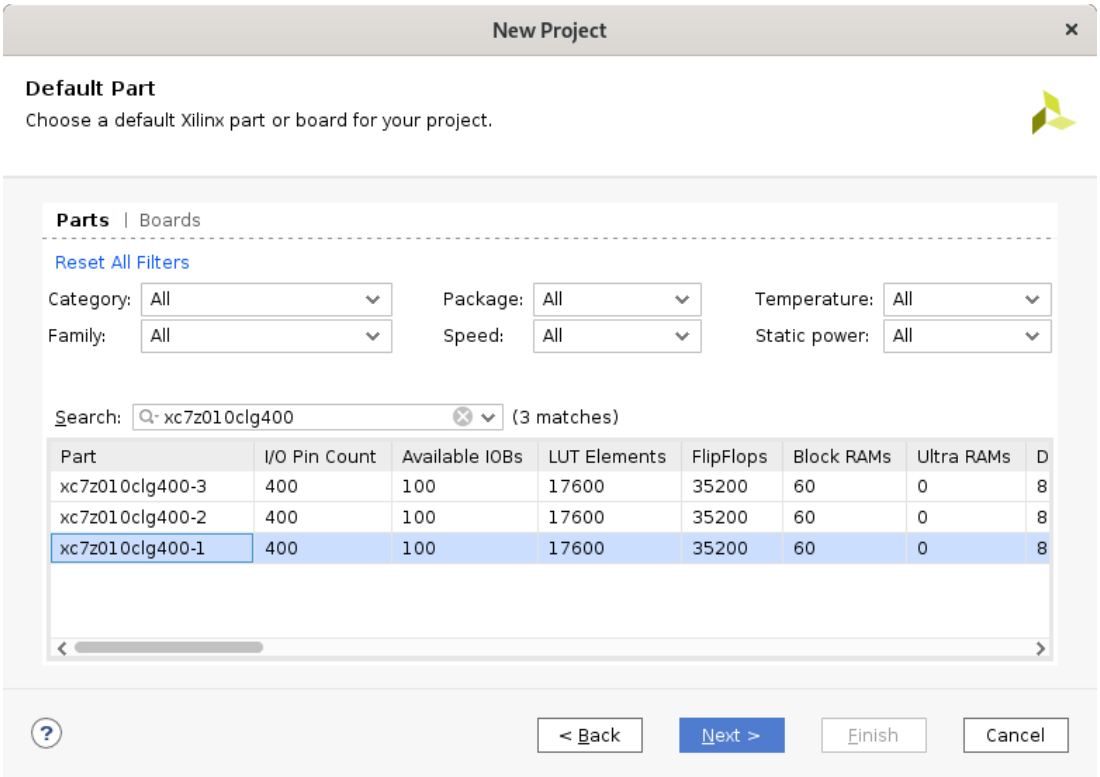

Figure 11 – Sélection du FPGA dans le projet Vivado

Question 4.6. Quel est le pourcentage d'utilisation des ressources ? (Figure 15). Commentez vos résultats.

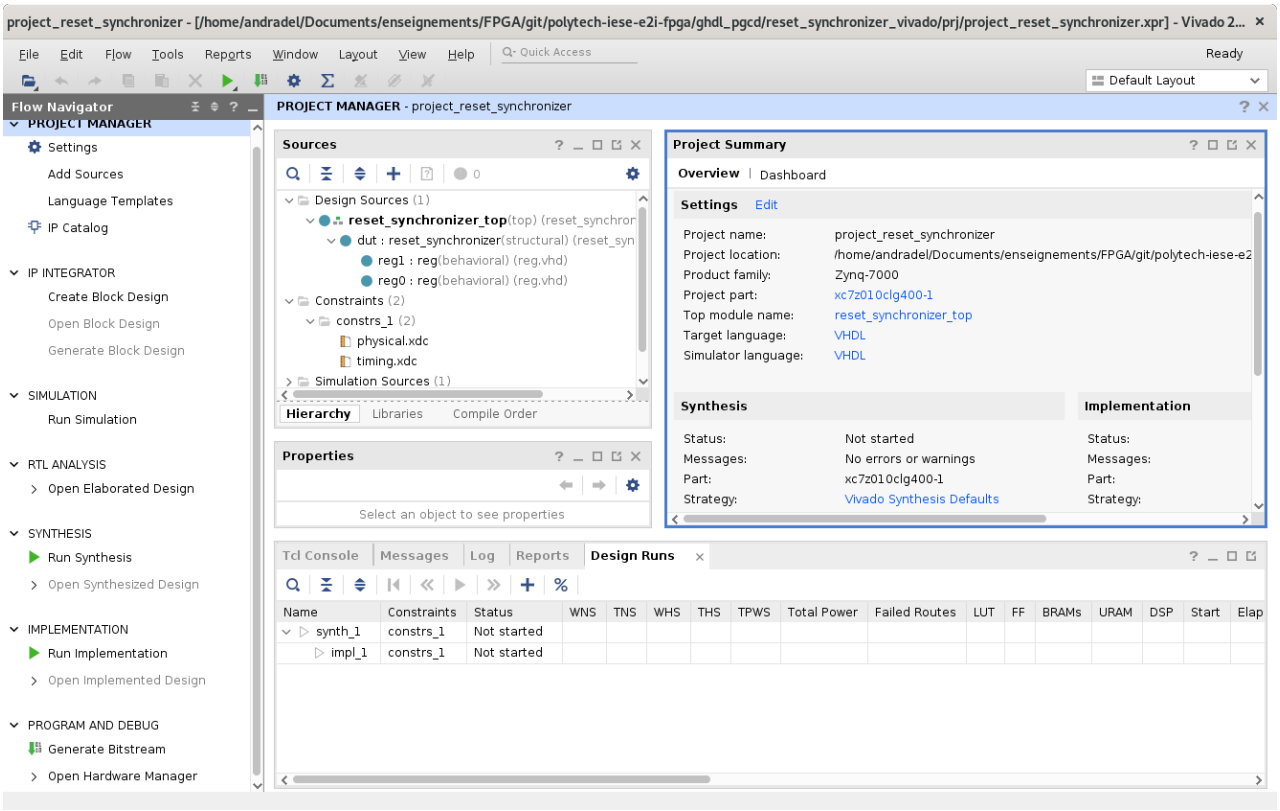

Figure 12 – Projet Vivado crée

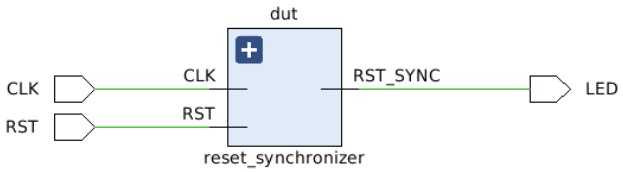

Figure 13 – Analyse RTL dans le projet Vivado

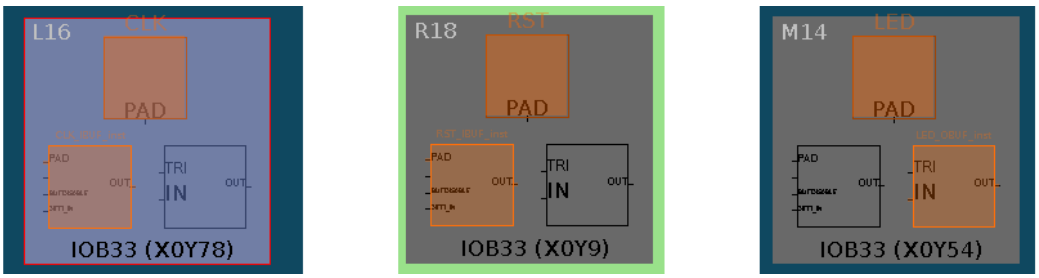

Figure 14 – Extraits du modèle synthétisé

4.4 Implémentation

— Allez dans `Implementation -> Run implementation` (choisissez les options par défaut)

— Attendez les résultats et ouvrez le modèle mis en œuvre. Utilisez le zoom pour retrouver les cellules utilisées. Commentez.

— Regardez aussi le report de “timing”. Vous devriez obtenir un résultat similaire à la Figure 16. Le WNS doit être toujours positif.

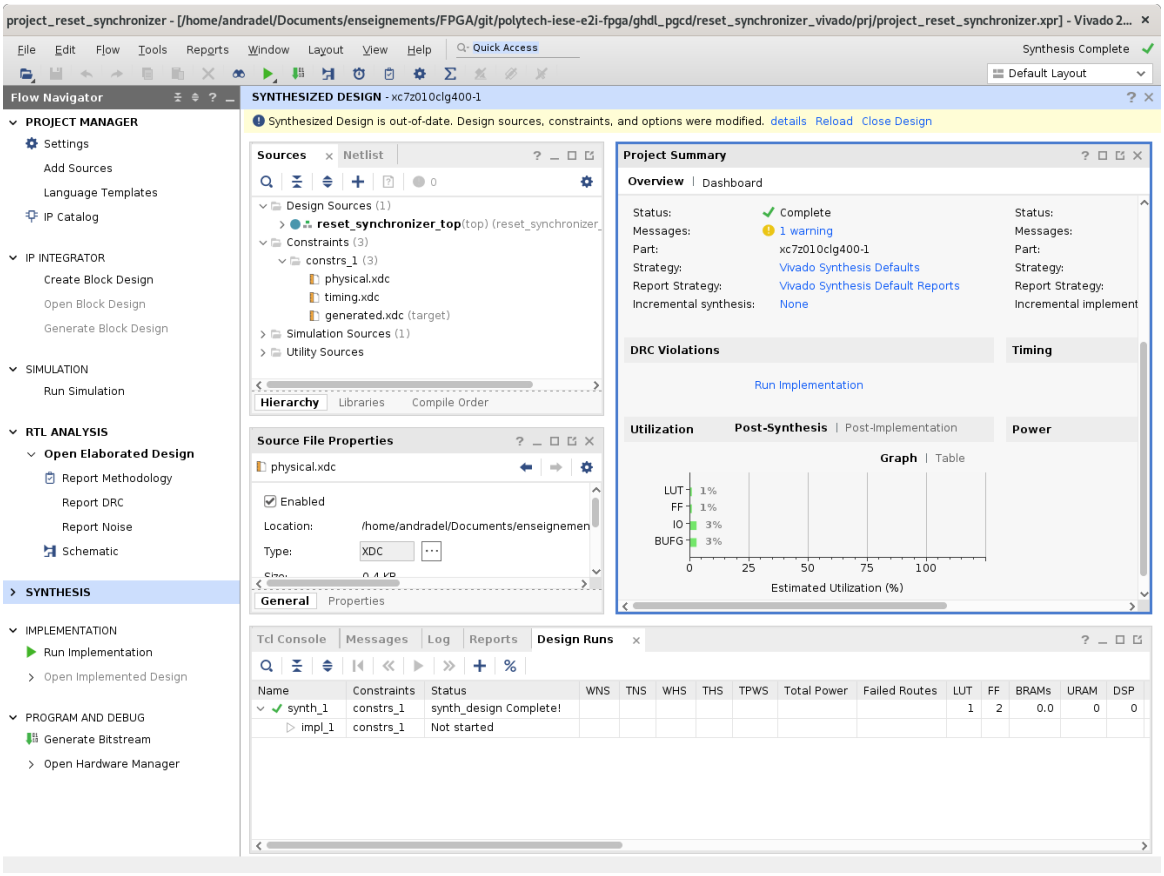

Figure 15 – Résultats de la synthèse

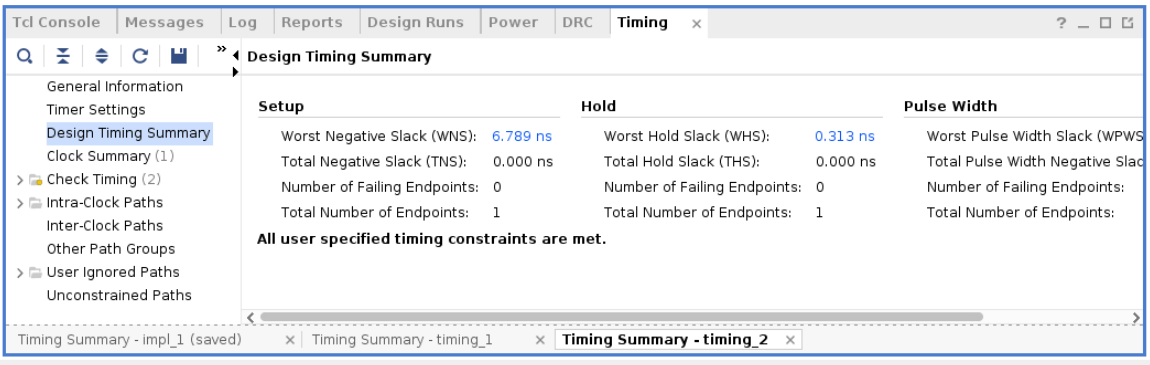

Figure 16 – Résultats de l'implémentation

Question 4.7. Le pourcentage d'utilisation des ressources a-t-il changé ? (Parfois l'outil fait des optimisations)

4.5 Programmation et debug

— Allez dans `Program and Debug -> Generate bitstream`

— Ouvrez `Open Hardware Manager -> Open Target -> Autoconnect`

  — Choisissez xc7z010-1 ou xc7z020-1 (selon votre FPGA)

  — Click droit `-> Program device` et selectionnez le `.bit`

— **Vérifiez sur carte!!!**

5 Mise en pratique : Analyse du code d'un synchroniseur générique

Dans cette partie vous devez analyser un modèle d'un synchroniseur générique de signal et réaliser son prototypage sur carte FPGA.

— Étudiez le code fourni dans le fichier `signal_synchronizer.vhd`

Question 5.1. Pouvez-vous expliquer le comportement ? Quel est l'utilité de `generic` dans ce modèle.

Question 5.2. Dessinez le circuit en question.

— Testez le circuit à l'aide du testbench fourni `signal_synchronizer_tb.vhd`

Question 5.3. Dans le `port map`, observez les valeurs de `INIT` et `D`. Pensez-vous qu'ils sont correctes ? Vous pouvez analyser le waveform généré pour confirmer votre réponse.

— Vous pouvez également créer un projet Vivado pour synthétiser ce composant et faire sont prototypage sur carte FPGA.

Question 5.4. Quelle conclusion pouvez vous obtenir après synthèse ? Quelles sont les différences avec le modèle synthétisé précédemment ?

6 Modélisation et prototypage FPGA d'un système complet : PGCD

Dans cette partie vous devez concevoir et implémenter sur FPGA un système complet permettant de faire le calcul du PGCD en suivant une approche PC/PO.

6.1 Vision globale du système et consignes

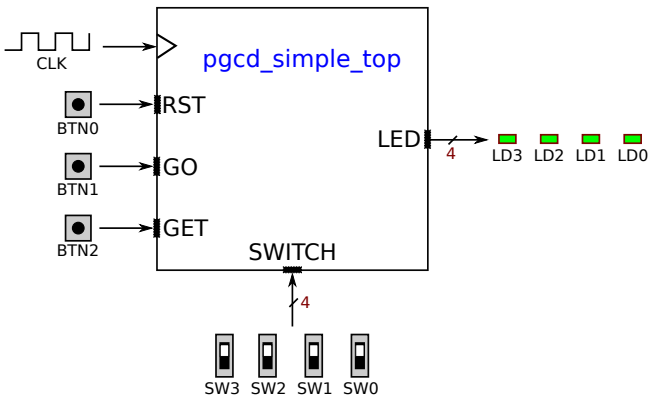

Figure 17 – Entité de plus haut niveau requise pour le prototypage sur carte FPGA

On propose de implémenter sur FPGA le modèle de la Figure 17. Ce composant aura le comportement suivant :

— **Étape 1 :** Sélectionnez les valeurs d'entrée avec les interrupteurs (switches). La valeur choisie avec les 4 interrupteurs permettra de générer différentes valeurs de `X` et `Y` pour le calcul du `PGCD(X,Y)`. Ce comportement sera implémenté dans une entité appelée `genxy` (à découvrir plus tard). Une fois la valeur choisie, appuyez sur le bouton poussoir `GO`.

— **Étape 2 :** Attendre l'allumage du `LED0` pour savoir que le PGCD a été calculé.

— **Étape 3 :** Appuyez sur le bouton poussoir `GET` (pour obtenir le résultat et l'envoyer sur les LEDs grâce au composant `genres`). Utilisez les interrupteurs à nouveau pour regarder le résultat (sur 32 bits).

6.2 Spécification de l'entité `PGCD` et validation par simulation

Pour la spécification de l'entité/architecture permettant de calculer le PGCD entre deux nombres de 32 bits `X` et `Y`, on vous propose de suivre l'approche PC/PO, dont la PO correspond au chemin de données (datapath) et la PC à la partie de contrôle (FSM) (voir Figure 18).

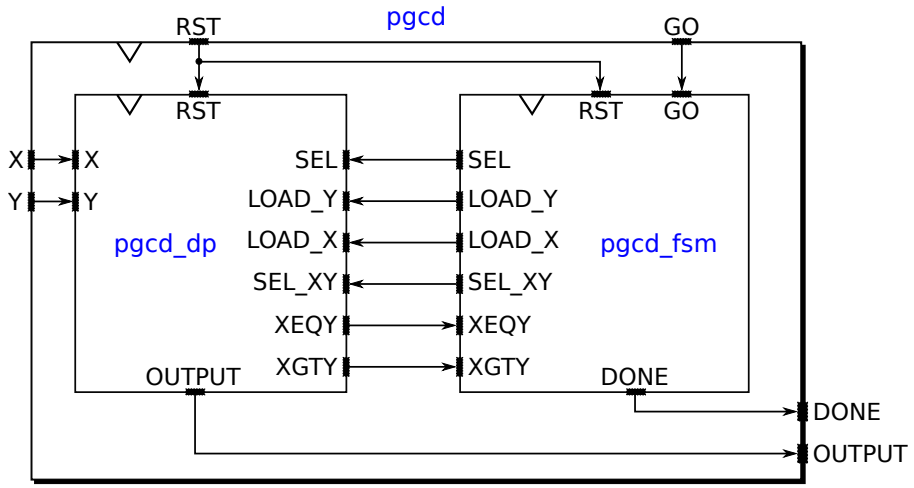

Figure 18 – Entité/architecture de l'entité PGCD

Le `PGCD` de `X` et `Y` est définie comme étant le plus grand diviseur commun de `X` et de `Y`. Il peut se calculer en utilisant l'algorithme présenté ci-dessous :

```
Entree X
Entree Y

while (X != Y) {
   if (X > Y) then
      X = X - Y
   else
      Y = Y - X
   end if
}

Sortie X
```

Avant de modéliser ce composant dans le fichier (`pgcd.vhd`), vous devez implémenter la PO et la PC en suivant les instructions présentées ci-dessous.

Attention : Une fois votre entité complètement implémentée, vous pourrez utiliser le banc de test fourni dans le fichier `pgcd_tb.vhd` pour valider votre composant.

6.2.1 Partie opérative (PO)

La partie opérative doit être définie dans le fichier `pgcd_dp.vhd`. Cette entité/architecture (Figure 19) contiendra les composants suivants (tous génériques sur `N` bits) :

— **muxn** : un multiplexeur 2 vers 1 permettant de choisir une des deux entrées `IN0` ou `IN1` (codées sur `N` bits) et la mettre sur la sortie `OUTPUT` en suivant un bit de sélection `SEL`.

— **regn** : un registre de `N` bits ayant une entrée d'horloge `CLK` de type `std_logic`, un signal asynchrone de reset `RST` actif haut, un signal `LOAD` de type `std_logic` actif haut qui permet de charger la valeur d'entrée sur la sortie, une entrée `D` de type `std_logic_vector` contenant la valeur à stocker et une sortie `Q` de type `std_logic_vector` qui sera initialisée à zéro pendant la phase de reset.

— **subn** : un soustracteur de deux nombres `A` et `B` (codés sur `N` bits). L'architecture du soustracteur sera construite grâce à l'instanciation de `N` soustracteurs de 1 bit. Pour l'instanciation vous utiliserez l'instruction **`for`** `... generate` et pour l'implémentation du soustracteur de 1 bit vous utiliserez les équations suivantes :

```
S = A xor B xor CIN
COUT = (not(A) and B) or (not(A) and CIN) or (CIN and B)
```

— **compn** : un comparateur qui prend deux nombres `X` et `Y` codés sur `N` bits en entrée et retourne comme sorties `XEQY` (qui prend la valeur `1` quand les deux entrées sont identiques et `0` dans le cas contraire) et `XGTY` (qui prend la valeur `1` quand l'entrée `X` est supérieur à l'entrée `Y` et `0` sinon).

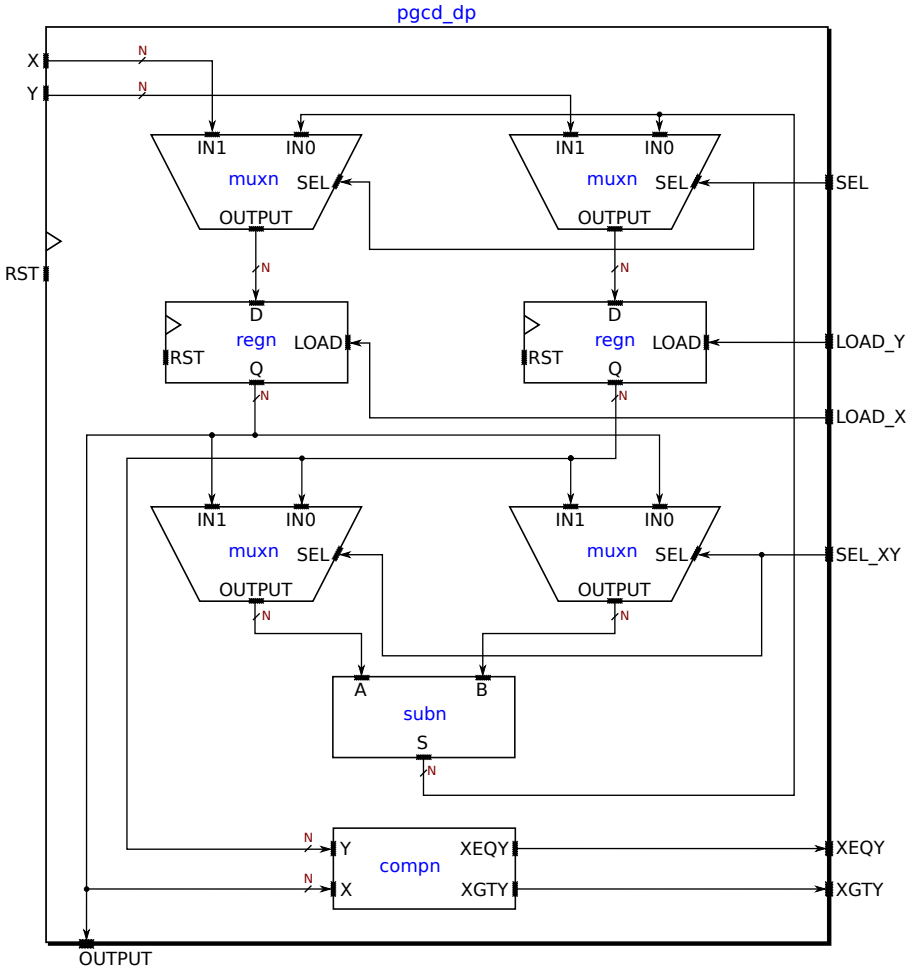

Figure 19 – Entité/architecture de la PO

Question 6.1. Écrivez l'entité/architecture de chaque composant selon les consignes données précédemment. Testez chaque composant de manière individuelle et validez son comportement par simulation. Tous les fichiers `.vdh` sont à rendre avec votre rapport (commentez votre code !).

Question 6.2. Définir en VHDL l'entité `pgcd_dp` et une architecture structurelle en respectant l'interface proposée dans la Figure 19. Construisez un banc de test et validez votre implémentation de calcul de PGCD avec des valeurs codées sur 32 bits. Expliquez les résultats obtenus.

Attention : Assurez-vous de bien comprendre le fonctionnement de la partie opérative avant de coder votre banc de test.

6.2.2 Partie contrôle (PC)

La partie contrôle doit être définie dans le fichier `pgcd_fsm.vhd`. Cette entité/architecture doit mettre en œuvre la machine à état (de type Moore) proposée dans la Figure 20.

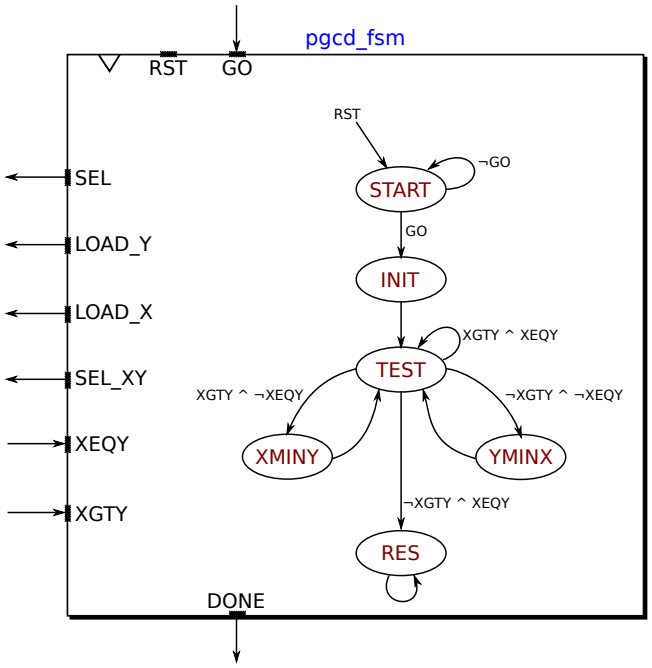

Figure 20 – Entité/architecture de la PC

Les sorties dépendront uniquement de l'état actuel selon le tableau présenté ci-dessous.

STATE/OUTPUT	SEL	LOAD_X	LOAD_Y	SEL_XY	DONE
START	0	0	0	0	0
INIT	1	1	1	0	0
TEST	0	0	0	0	0
XMINY	0	1	0	1	0
YMINX	0	0	1	0	0
RES	0	0	0	0	1

Question 6.3. Définir en VHDL l'entité `pgcd_fsm` et une architecture fsm en respectant l'interface proposée dans la Figure 20. Construisez un banc de test et validez votre implémentation. Expliquez les résultats obtenus.

6.2.3 Assemblage PC/PO

Question 6.4. En utilisant comme référence l'entité/architecture proposées dans la Figure 18, assemblez la machine d'état et le chemin de données pour faire un seul circuit. Testez votre composant à l'aide du banc de test fourni et commentez vos résultats.

6.3 Prototypage FPGA du système

En suivant la même procédure proposée dans la Section 4, implémentez votre PGCD sur votre carte FPGA. Vous devrez d'abord créer l'entité/architecture de plus haut niveau `pgcd_simple_top` et ensuite définir les fichiers des contraintes, créer votre projet Vivado, effectuer la synthèse, l'implémentation et la programmation sur carte.

Attention : Le fichier `pgcd_simple_top` est incomplet. Vous devez le compléter en suivant l'architecture proposée dans la Figure 21 et le déplacer dans le répertoire `src` du dossier `pgcd_simple_top_vivado` que vous devrez aussi créer.

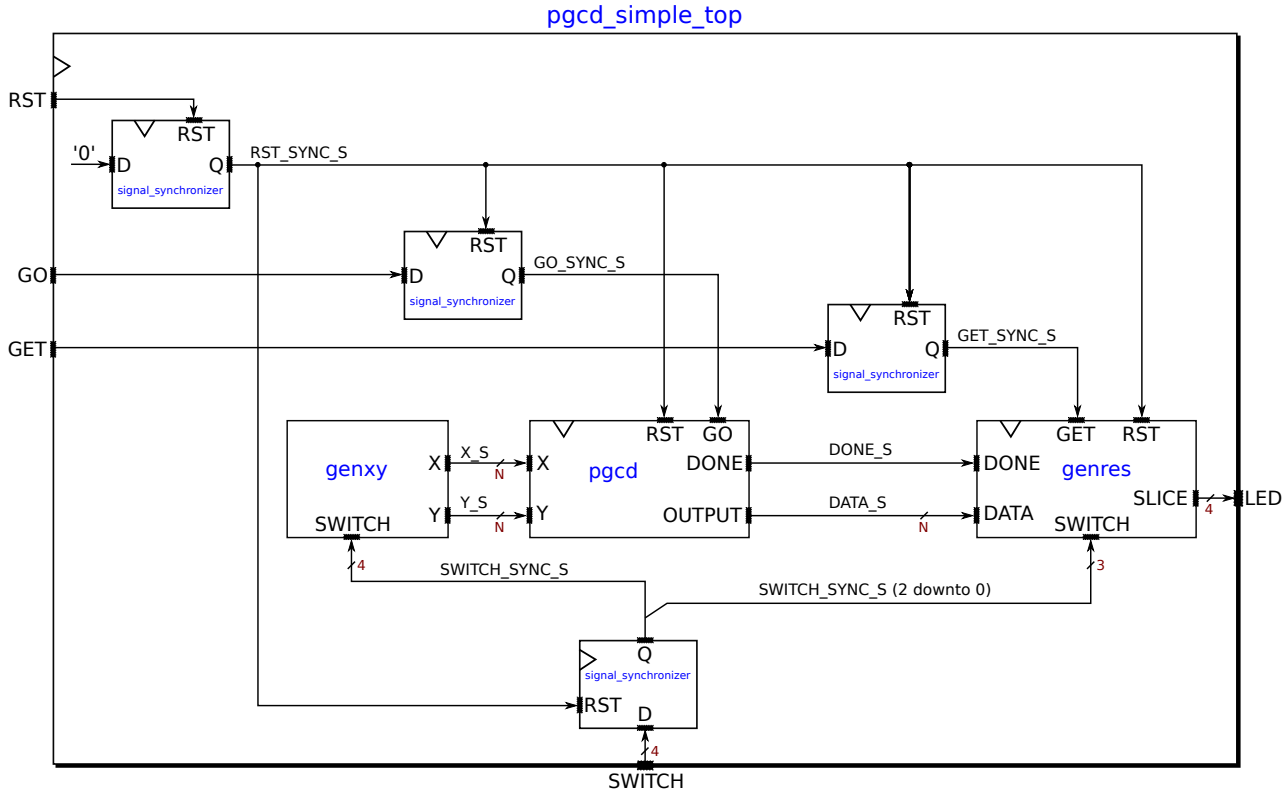

Figure 21 – Architecture de l'entité de plus haut niveau (PGCD)

Tous les composants internes (sauf le `pgcd`) sont déjà définis et prêts à être instanciés dans votre modèle.

Question 6.5. Créez votre projet Vivado et implémentez le système complet sur votre carte FPGA. Commentez les résultats obtenus à chaque étape, ajoutez dans votre rapport les schémas générés par Vivado et commentez les résultats. Testez sur carte le comportement souhaité et décrit dans la Section 6.1.

6.4 Pour aller plus loin ...

Proposez un nouveau système (par exemple, le calcul de la factorielle d'un nombre !)

7 Le rapport final

Envoyer le rapport final en format `pdf` à `liliana.andrade@univ-grenoble-alpes.fr`. Ce rapport doit être envoyé **au plus tard** le mardi 12 janvier 2021.

Fichiers à rendre :

— Tous les fichiers `.vhd` des entités/architectures et bancs de test de chaque composant utilisé dans votre modèle. Ces fichiers doivent être bien commentés et indiquer vos noms et prénoms (en commentaire).

— Le rapport avec les captures d'écran des vos résultats de simulation et de synthèse. Ainsi que l'explication de chaque étape mise en œuvre.