POLYTECHNIQUE MONTRÉAL

UNIVERSITÉ D'INGÉNIERIE

INF3500 - Conception et réalisation de systèmes numériques

Laboratoire 4

Nicolas Deloumeau
Robin Gay

Automne 2024

Le calcul de la racine carré par la méthode de Newton pipelinée

À la fin de ce laboratoire, vous devrez être capable de :

- Concevoir et modéliser en VHDL un chemin des données qui réalise des fonctions arithmétiques et logiques complexes au niveau du transfert entre registres (RTL : *Register Transfer Level*). (B5)
  - Instancier des registres, dont des compteurs
  - Implémenter les fonctions arithmétiques et logiques pour les valeurs des registres, correspondant à une spécification par micro-opérations ou par pseudocode
- Composer un banc d'essai pour stimuler un modèle VHDL d'un chemin des données. (B5)
  - Générer un signal d'horloge et un signal de réinitialisation
  - Générer et appliquer des stimuli à un circuit séquentiel
  - Comparer les sorties du circuit à des réponses attendues pré-calculées
  - Utiliser des énoncés `assert` ou des conditions pour vérifier le module
  - Générer un chronogramme résultant de l'exécution du banc d'essai, et l'utiliser pour déboguer le circuit, entre autres pour résoudre les problèmes de synchronisation
- Implémenter un chemin des données sur un FPGA
  - Effectuer la synthèse et l'implémentation du circuit
  - Extraire, analyser et interpréter des métriques de coût d'implémentation
  - Programmer le FPGA et vérifier le fonctionnement correct du circuit avec l'interface PYNQ

Ce laboratoire s'appuie sur et complémente le matériel suivant :

1. Les procédures utilisées et les habiletés développées dans les laboratoires 0, 1 et 2
2. Le calcul de la racine carré dans le laboratoire 3
3. La matière des cours des semaines 11 (Pipelinage).

Liste des fichiers

Fichier	Contenu
`src/division_par_reciproque.vhd`	Définit le module de division par réciproque utilisé au labo précédent
`src/racine_carree.vhd`	Contient le module de calcul de racine carré à pipeliner
`src/top.vhd`	*Top-level* du projet
`xdc/PYNQ-Z2-v1.0.xdc`	Fichier de contraintes pour la PYNQ-Z2
`synth-impl/create_zynq_wrapper.tcl` `synth-impl/zynq.tcl`	Scripts de génération du *wrapper* pour le Zynq, à sourcer après avoir importé les sources

Partie 0 : Introduction

Le pipelinage

Le pipelinage consiste à découper un circuit en étages séparés par des registres. Il sert deux objectifs principaux :

- augmenter le débit de données du circuit en exploitant un parallélisme partiel
- augmenter la fréquence d'opération maximale du circuit en réduisant la longueur des chemins critiques combinatoires

Dans ce laboratoire, nous allons nous intéresser au premier point. En partant du module de racine carré du laboratoire précédent, vous allez dérouler la boucle dans l'espace. Ainsi, le nombre d'étages de la pipeline sera donné par `kmax` + 1, où `kmax` est le nombre d'itérations de la division à réaliser.

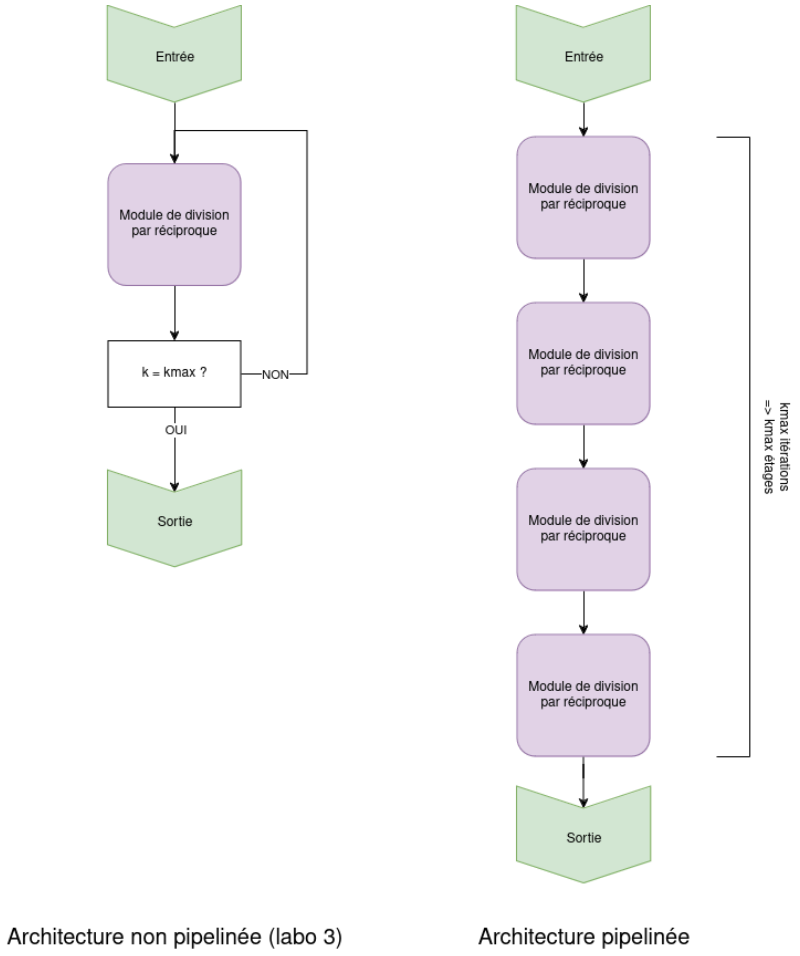

Fig. 1. – Architecture non pipelinée vs. architecture pipelinée

> ⚠ **Attention**
>
> Dans ce laboratoire, nous travaillons avec un `kmax` très réduit (`kmax` = 2). Par conséquent, les valeurs en sortie ne seront probablement pas proches des racines carré recherchées : il est normal d'obtenir 63 pour tous les nombres. Pour obtenir un design pipeliné et avec des résultats corrects, il faudrait une dizaine d'étages de pipeline, mais cela risque de crasher à la synthèse car Vivado aurait besoin de beaucoup de mémoire RAM pour synthétiser un tel circuit.

Le protocole AXI-Stream

Dans cet exercice de laboratoire, on reprend le problème du calcul de la racine carré développé précédament. L'objectif est de pipeliner le calcul, pour effectuer un grand nombre d'opérations. Cela permet de calculer la racine carré d'un grand nombre d'entrées avec un débit intéressant.

Pour communiquer avec la platforme PYNQ, le protocole AXI-Stream sera utilisé. Il va permettre d'envoyer une séquance de nombres dont on veut connaître la racine carré, et de retourner la séquence de résultats au processeur après calcul.

Le protocole AXI-Steam contient 4 signaux principaux:

- `data` : donnée
- `valid` : indique qu'une donnée est sur le bus
- `last` : indique la dernière donnée d'une séquence
- `ready` : le receveur est prêt à recevoir les données

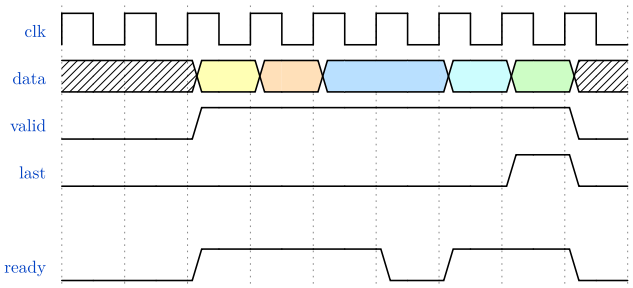

Fig. 2. – Exemple d'une séquence AXI-Stream

Pour des raisons de simplification, le signal `ready` du port AXI-Stream d'entrée est relié au signal `ready` du port AXI-Stream de sortie. (Visible dans `top.vhd`). Il est donc important de mettre en pause la pipeline, sinon des données pourrait être perdues lorsque le processeur ZYNQ n'est pas prêt à en recevoir. Donc, puisque la pipeline est en pause, il n'est pas possible de recevoir de données et donc le signal `ready` à l'entrée a la même valeur.

Implémentation et validation sur la carte

> ⚠ **Attention**
>
> Comme nous allons utiliser l'accès mémoire en AXI-Stream, nous ne pouvons pas simplement programmer la carte avec Vivado comme nous l'avons fait dans les labos précédents. Réalisez les étapes suivantes pour pouvoir programmer votre design sur votre FPGA.

Une fois votre *bitstream* généré, connectez-vous à l'interface Jupyter. Dans un **nouveau dossier** :

- Ajoutez le *bitstream*, généré à `<projet_vivado>/<projet>.runs/impl_1/top.bit`.
- Ajoutez le fichier `.hwh` que vous trouverez à `<projet_vivado>/<projet>.gen/sources_1/bd/zynq/hw_handoff/zynq.hwh` et renommez-le `top.hwh`.
- Créez un nouveau notebook et ajoutez-y le code 1. N'hésitez pas à séparer le code sur plusieurs cellules pour ne pas tout relancer à chaque test.

```python
# Import des librairies
import numpy as np
from pynq import allocate, Overlay

# Programmer le bitstream
ol = Overlay("top.bit")

# Allocation des buffers
ds = 4000
ibuf = allocate(shape=(ds,), dtype=np.uint32)
obuf = allocate(shape=(ds,), dtype=np.uint32)
for i in range(ds):
    ibuf[i] = i + 1
    # Vérification des valeurs
    if i < 10:
        print(f"{ibuf[i]},", end=' ')
print("\n---")

# Transfert des données vers le module AXI-Stream
ol.axi_dma_0.sendchannel.transfer(ibuf)
# Réception des résultats
ol.axi_dma_0.recvchannel.transfer(obuf)

# Afficher le contenu du buffer
for i in range(10):
    print(f"{obuf[i]},", end=' ')
```

Code 1. – Code à ajouter au *notebook* Python

Si les données sont correctement passées à travers le pipeline, vous devriez voir les résultats affichés en sortie.

Partie 1 : conception du module de racine carré pipeliné

Complétez le design du module de calcul de la racine carré pipeliné donné dans le fichier `racine_carree_pipeline.vhd` pour implémenter une pipeline à **3 étages (`kmax` = 2)**. Pour ce faire, vous devrez instancier **un diviseur par réciproque** pour chaque itération, soit 2 au total. Utilisez des `array` pour clarifier vos signaux. Vous pouvez créer de nouveaux types en prenant exemple sur ceux implémentés dans le code source.

> ⚠ **Attention**
>
> Pour les besoins de ce laboratoire, vous devez choisir `N` = 16, `M` = 8 et `W_frac` = 14.

> *i* **Info**
>
> Rappelez-vous qu'il est essentiel, dans un chemin de données pipeliné, que les données arrivent toutes en même temps. Assurez-vous que les données d'entrée, les signaux `valid` et `last` soient bien synchronisés jusqu'à la sortie.

> ! **Évaluation**
>
> Remettez votre fichier `racine_carree_pipeline.vhd` modifié dans votre dépôt Git. *N'oubliez pas de* push.

Partie 2 : banc d'essai

Vérifiez le fonctionnement de votre module `racine_carree_pipeline.vhd` à l'aide du banc d'essai du fichier `racine_carree_pipeline_tb.vhd`. Pour ce laboratoire, pas besoin de réaliser un test exhaustif : choisissez judicieusement quelques valeurs à tester, passez-les à votre module et vérifiez ses sorties avec des énoncés `assert`.

Attention, pour passer des données à votre module, vous devrez implémenter une transaction AXI-Stream dans votre banc d'essai. Veillez à bien avoir compris l'introduction à ce sujet pour pouvoir tester correctement votre module.

> ! **Évaluation**
>
> Remettez votre banc d'essai modifié avec vos stimuli sur votre dépôt Git. Ajoutez aussi une capture de simulation et justifiez sommairement votre choix de vecteurs de test.

Partie 3 : Implémentation et vérification

Une fois le module décrit et testé, implémentez-le sous forme d'un *bitstream* et flashez-le sur votre carte, comme expliqué dans la partie 0. Testez-le avec un *notebook* Python.

Notez qu'il est normal que vous obteniez 63 sur pour toutes les entrées. Avec `kmax` = 2, le calcul n'a pas assez d'itérations pour être correct.

> **! Évaluation**
>
> Remettez votre fichier `top.bit` sous `synth-impl`, et le *notebook* Python que vous avez utilisé pour le tester à la racine du dépôt.

Partie 4 : Défi

Mise en garde. *Compléter correctement les parties précédentes peut donner une note de 17 / 20 (85%), ce qui peut normalement être interprété comme un A. La partie 4 demande du travail supplémentaire qui sort normalement des attentes du cours. Il n'est pas nécessaire de la compléter pour réussir le cours ni pour obtenir une bonne note. Il n'est pas recommandé de s'y attaquer si vous éprouvez des difficultés dans un autre cours. La partie 4 propose un défi supplémentaire pour les personnes qui souhaitent s'investir davantage dans le cours INF3500 en toute connaissance de cause.*

4a : Implémentation à `kmax` variable

Dans cette partie, nous vous demandons de rendre votre design plus souple en implémentant une pipeline à `kmax` étages. Pour ce faire, vous allez devoir **ajouter un `generic`**, `kmax`, à la liste de `generic` de votre module `racine_carre_pipeline` et rendre la génération des modules de division conditionnelle à la valeur de `kmax`. Vous devrez utiliser une **boucle de génération** en VHDL pour instancier `kmax` modules de division dans votre `racine_carre_pipeline.vhd`.

Reproduisez les résultats du laboratoire précédent en générant un *bitstream* pipeliné avec **`kmax` = 11**. Validez les résultats obtenus avec votre *notebook*.

⚠ Attention

Il est possible que Vivado consomme énormément de mémoire vive à la génération du *bitstream*, selon la configuration des limites mémoire de votre OS et la quantité de RAM de votre machine. Si le processus fait planter votre ordinateur, ou est vraiment trop long, vous pouvez baisser `kmax` jusqu'à obtenir une génération faisable. **Dans tous les cas, assurez-vous d'avoir tout sauvegardé avant de lancer la génération du *bitstream* pour cette partie**. *Notez que nous avons réussi à le générer avec 16GB de RAM et sans swap-space.*

Si vous avez confiance dans les capacités de votre machine et de votre OS, vous pouvez augmenter `kmax` pour le fun.

! Évaluation

Remettez votre *bitstream* avec `kmax` = 11 (si vous avez dû diminuer `kmax`, indiquez-le dans votre rapport), et vos fichiers `racine_carree_pipeline.vhd` et `racine_carree_pipeline_tb.vhd` modifiés.

4b : Pourquoi j'ai fait tout ça ?

! Évaluation

- Calculez le **temps nécessaire** pour obtenir **10000 résultats** avec le module **non pipeliné** du laboratoire 3 **en fonction de `kmax` et de `f_clk`**, la fréquence de l'horloge.
- Calculez le **temps nécessaire** pour obtenir **10000 résultats** avec le module **pipeliné** de ce laboratoire **en fonction de `kmax` et de `f_clk`**, la fréquence de l'horloge.
- Donnez les **valeurs numériques en secondes** de ces deux résultats en supposant que `f_clk` = 100 MHz et `kmax` = 11. **Expliquez** en quoi ce résultat montre l'utilité du pipelinage.

Remise

La remise se fait directement sur votre dépôt Git. Faites un *push* régulier de vos modifications, et faites un *push* final avant la date limite de la remise. Respectez l'arborescence de fichiers originale. Consultez le barème de correction pour la liste des fichiers à remettre.

Directives spéciales :

- Ne modifiez pas les noms des fichiers, entités ou architectures, ni les listes de `generics` ou de ports
- Remettez du code de très bonne qualité, lisible, bien indenté et commenté.
- Indiquez clairement la source de tout code que vous réutilisez ou dont vous vous inspirez.

Barème de correction

Le barème de correction est progressif. Il est relativement facile d'obtenir une note de passage (> 10) au laboratoire et il faut mettre du travail pour obtenir l'équivalent d'un A (17/20). Obtenir une note plus élevée (jusqu'à 20/20) nécessite plus de travail que ce qui est normalement demandé dans le cadre du cours et plus que les 9 heures que vous devez normalement passer par semaine sur ce cours.

Critères	**Points**
Partie 1 • Votre module final `racine_carree_pipeline.vhd`	6
Partie 2 • Votre banc d'essai complet dans le fichier `racine_carree_pipeline_tb.vhd` • Votre capture de simulation • Vos vecteurs de test	7
Partie 3 • Votre fichier `top.bit` fonctionnel • Votre *notebook* Python	2
Général • Qualité, lisibilité et élégance du code : alignement, choix des identificateurs, qualité et pertinence des commentaires, respect des consignes de remise incluant les noms des fichiers, orthographe, etc.	2
Sous-total	**17**
Partie 4 • 4a : Généralisation du module avec une boucle de génération et un `generic` • 4b : Calculs et discussion	2 1
Total	**20**