Compte rendu TP A3
Système électronique embarqué : Nios

GHAOUI M. Anis

25 - Mars - 2020

Contents

1 Introduction **2**

2 Conception d'opérateur racine carrée **2**
- 2.1 Implémentation combinatoire . . . 2
- 2.2 Implémentation multi-cycles . . . 3
  - 2.2.1 4 cycles . . . 3
  - 2.2.2 9 cycles . . . 4
  - 2.2.3 Variante "avare" . . . 4
- 2.3 Implémentation avec opérateur unique . . . 5
  - 2.3.1 Opérandes préemptées . . . 5
  - 2.3.2 Ségrégation des machines d'états . . . 5
- 2.4 Implémentation Pipeline . . . 6

3 Comparaison et résultat **6**

4 Conception du système embarqué **8**
- 4.1 Implémentation et programmation du Nios . . . 8
- 4.2 Usage des différentes mémoires . . . 8
- 4.3 Comparaison des différentes implémentations . . . 8
  - 4.3.1 Mesure de temps . . . 9
  - 4.3.2 Temps mesurés . . . 9
- 4.4 Intégration du coprocesseur . . . 10
  - 4.4.1 Entrées/Sorties . . . 10
- 4.5 Intégration des instructions personnalisées . . . 11
- 4.6 Comparaison . . . 11

5 Conclusion **12**

1 Introduction

La complexité de la conception des systèmes embarqués modernes étant devenue trop élevée, il est indispensable de recourir à des outils afin d'automatiser ce processus fastidieux. Dans le cours A3, on voit que les étapes de cette conception consistent en la préparation d'un système ayant un CPU virtuel dit *SoftCore*, une ou plusieurs mémoire pour accompagner ce CPU et une brique FPGA qui agit comme un accélérateur matériel.

Durant les séances de TP, on procède à la conception de plusieurs variante cette brique afin d'effectuer le calcul d'une racine carrée entière sur FPGA décrite en VHDL. Puis, on instancie un système embarqué grâce à l'outil Qsys d'Intel/Alter pour avoir un CPU et les périphériques requis. Enfin, on intégrera la brique conçue dans ce système afin de mesurer ses performances globales.

Tous les codes et les projet Quartus peuvent être trouvés sur ce lien.

2 Conception d'opérateur racine carrée

On commence par une analyse de l'algorithme afin de comprendre sa complexité et l'utilité d'accélérer un tel calcul matériellement.

Input: (X,n) : X entier codé sur $2 \times n$ bits
Output: Z : Z entier codé sur n bits

```
1  Charger X
2  V = 2^(2n-2)
3  Z = 0
4  for i = n-1 à 0 do
5  |   Z = Z + V
6  |   if X − Z ≥ 0 then
7  |   |   X = X − Z
8  |   |   Z = Z + V
9  |   else
10 |   |   Z = Z − V
11 |   Z = Z/2
12 |   V = V/4
13 retourner Z
```

2.1 Implémentation combinatoire

Dans un premier temps, on pense à simplement traduire **tout** l'algorithme en vhdl dans un seul *process* afin d'avoir une séquence d'éléments combinatoires propageant le résultat pour chaque valeur de i. Ceci sera synthétisé comme un circuit volumineux.

Résultats On voit sur la figure 1 le lancement du calcul sur le circuit combinatoire. celui ci est entre 2 coups d'horloges en simulation. Il sera contraint par les limites matérielles sur la carte.

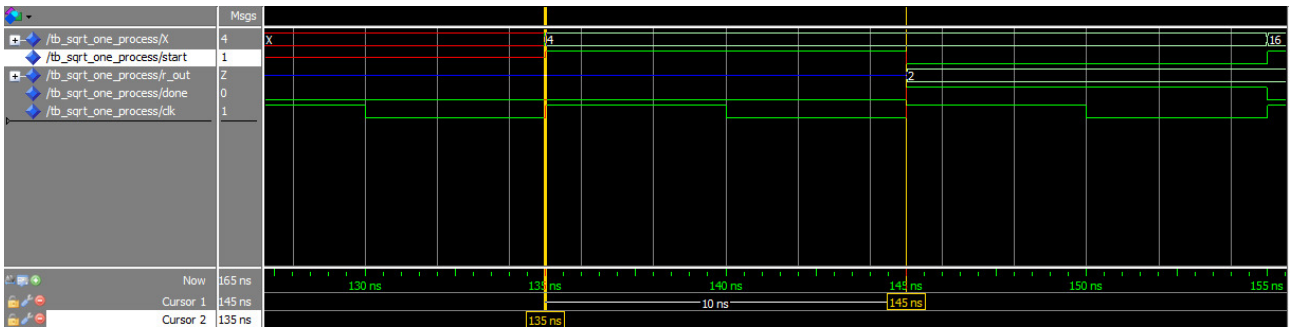
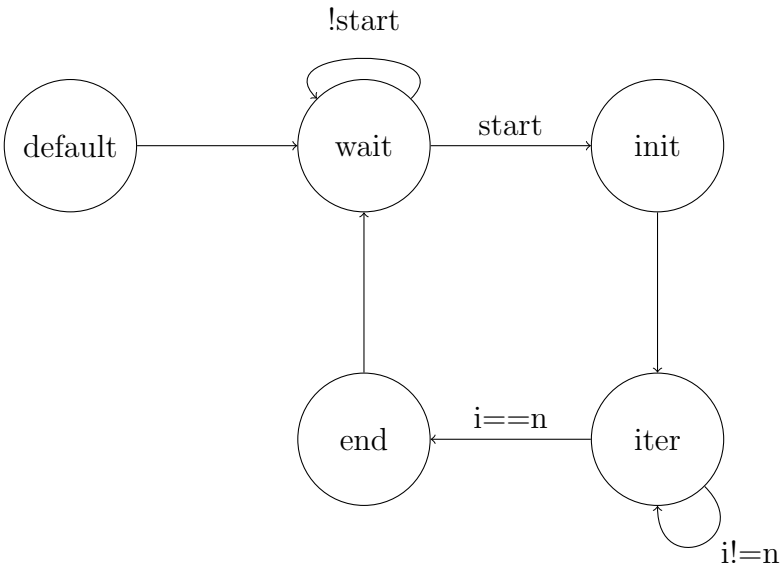


Figure 1: Diagramme temporel du one process

2.2 Implémentation multi-cycles

2.2.1 4 cycles

On choisit alors d'implémenter une machine synchrone où on décrit le module de racine carrée par une machine d'états finis. il y aura un cycle où on itère n fois . Le diagramme ci-dessous représente cet automate :



Résultats On voit sur la figure 2 le lancement du calcul sur le circuit synchrone. On a alors une latence observable contrairement à celui en combinatoire.

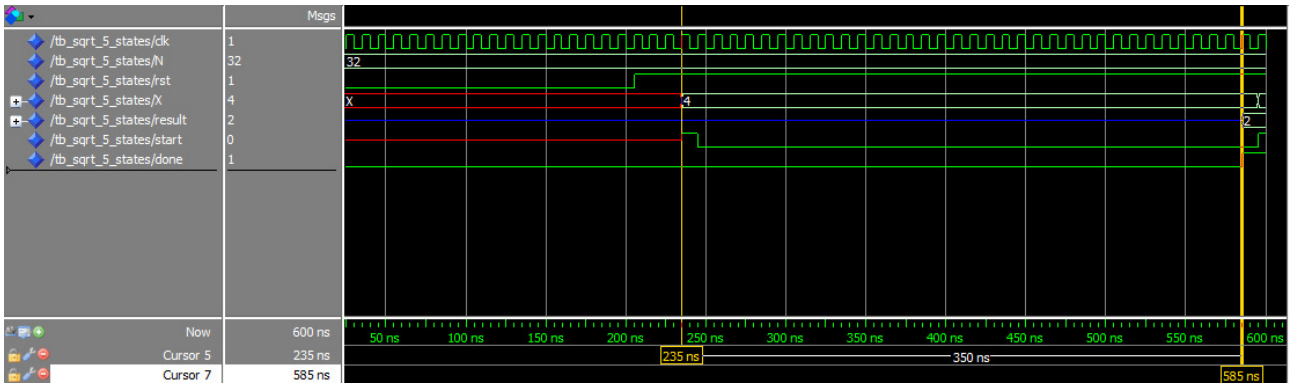
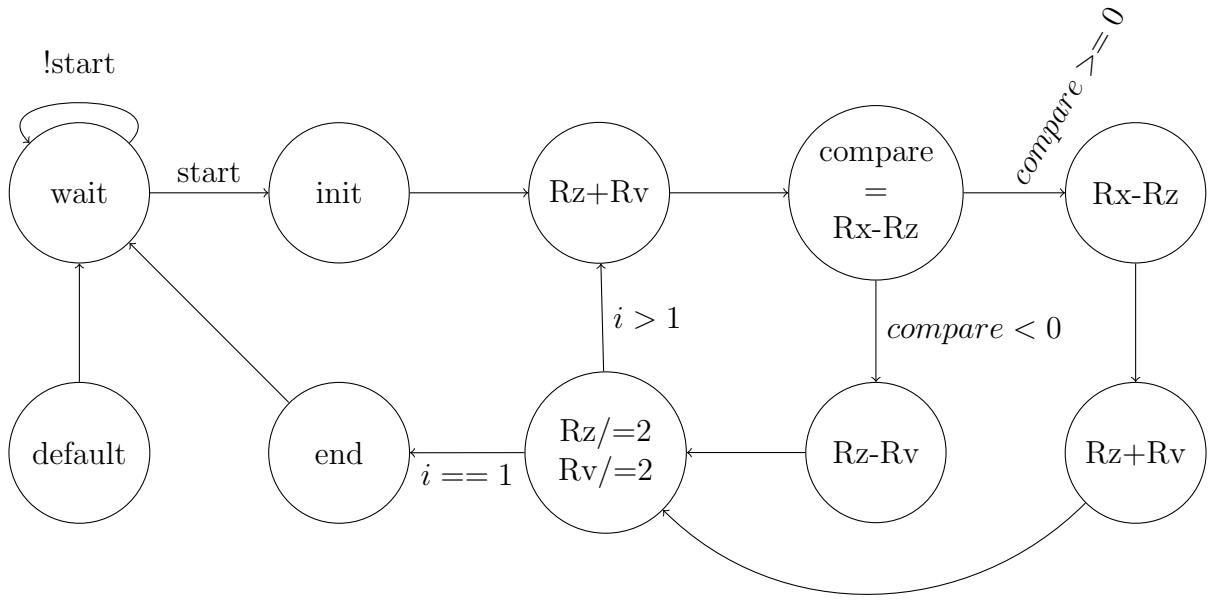


Figure 2: Diagramme temporel du 5 states

2.2.2 9 cycles

Cette implémentation décompose relativement bien les états de l'automate. Mais, on pense pouvoir obtenir un gain de performances en ayant un cycle par opération à effectuer. L'idée est que vu que l'état sera combinatoirement simple, il sera plus rapide à exécuter et donc la fréquence augmente. Par contre, on pense aussi que ceci consommera plus de ressources du FPGA.



Résultats On voit sur la figure 3 le lancement du calcul sur le circuit synchrone. On a plus de cycles de latence mais on pense que la fréquence de fonctionnement devrait être plus importante.

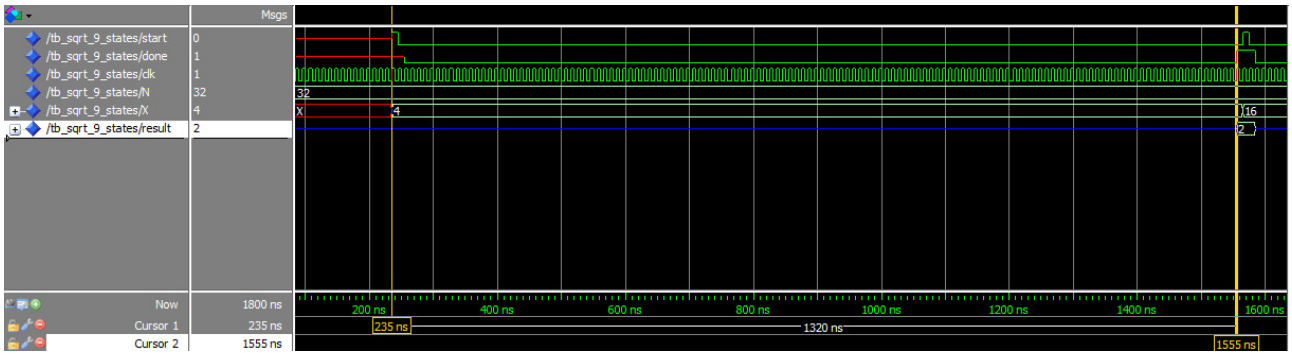


Figure 3: Diagramme temporel du 9 states

2.2.3 Variante "avare"

On procède a une variante dite "avare" qui va exploiter au mieux les mécanismes VHDL. Celle-ci parait offrir un compromis entre surface et fréquence.

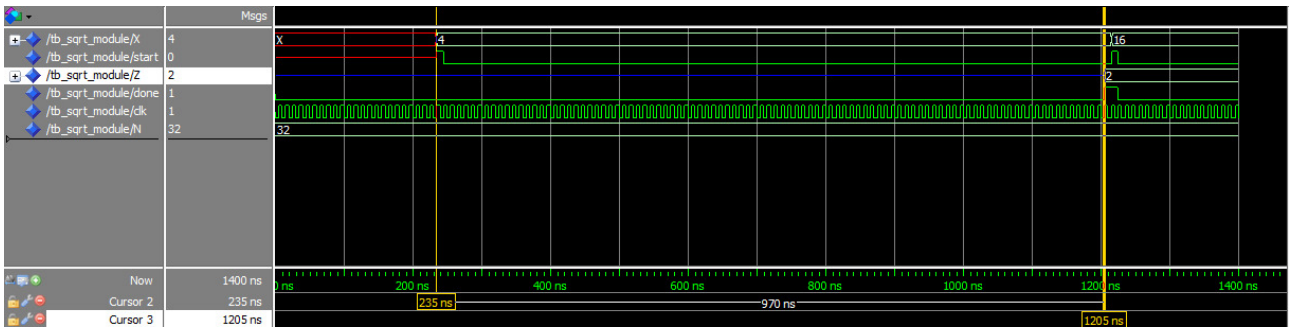


Figure 4: Diagramme temporel de sqrt

2.3 Implémentation avec opérateur unique

On propose alors une implémentation avec un seul addition/soustracteur proposé dans l'énoncé. Ceci a pour but de diminuer le nombre d'éléments logiques utilisés. Il existe plusieurs implémentations de cet opérateur qui ont toutes comme contraintes de n'instancier qu'un seul opérateur qui est fourni par l'énoncé.

2.3.1 Opérandes préemptées

Dans cette variante, on positionne les variables à l'entrée de l'opérateur à un cycle avant leur utilisation. Ceci va contraindre la synthèse à avoir une période d'horloge plus grande que la durée de l'opération combinatoire d'addition/soustraction.

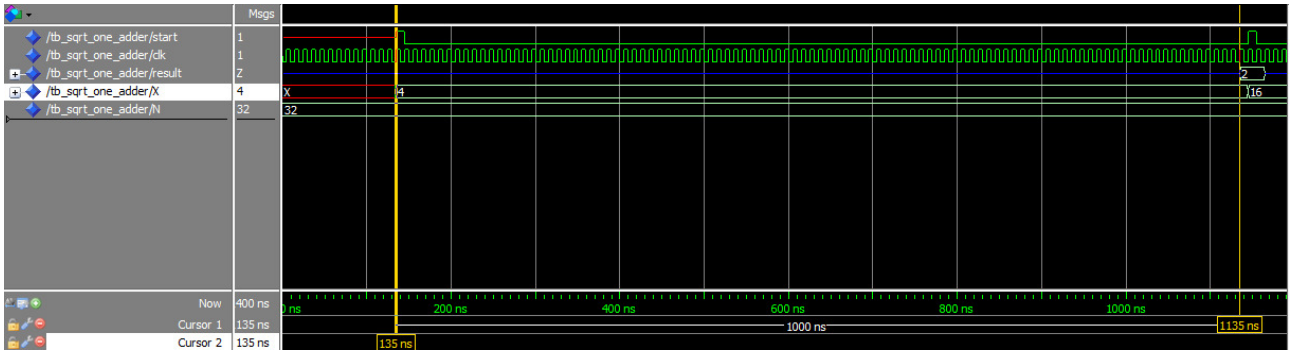


Figure 5: Diagramme temporel du one adder

On constate sur la figure 5 que le nombre de cycle est conséquent. On s'attend alors à ce que l'occupation matérielle soit moindre. On aura 2 versions de cette implémentation: l'une utilisant simplement un registre i pour compter, l'autre utilisant le décompteur fourni.

2.3.2 Ségrégation des machines d'états

On procède à une séparation des machines d'états. On distingue la logique de contrôle synchrone de la logique de calcul qui est combinatoire. Ainsi, on aura plus de cycles mais beaucoup plus rapide et une consommation minime des ressources FPGA.

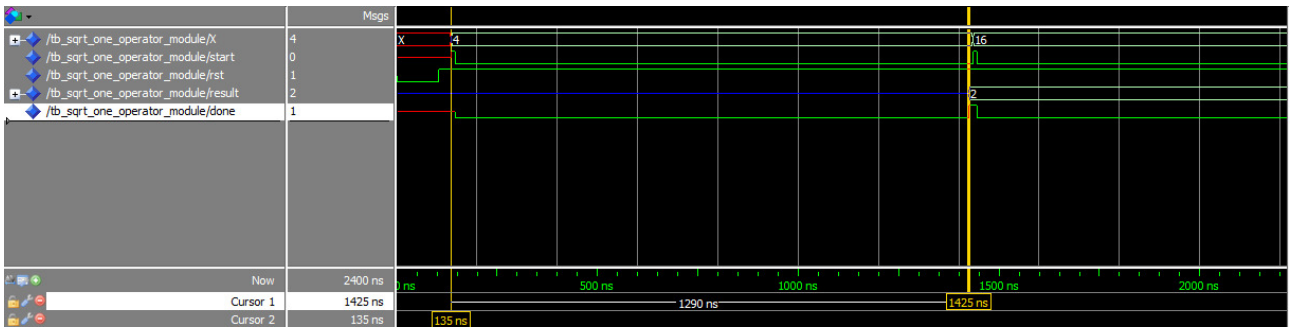


Figure 6: Diagramme temporel de one operator

2.4 Implémentation Pipeline

Enfin, on pense à l'implémentation du module racine carrée en mode pipeline. On vise à ce que ce pipeline ait un intervalle d'initialisation de 1. i.e. qu'il puisse absorber une donnée à chaque nouveau cycle. Il aura naturellement une latence égale à n (nombre de bit de l'entrée) car chaque itération de la boucle d'algorithme est traité dans un étage.

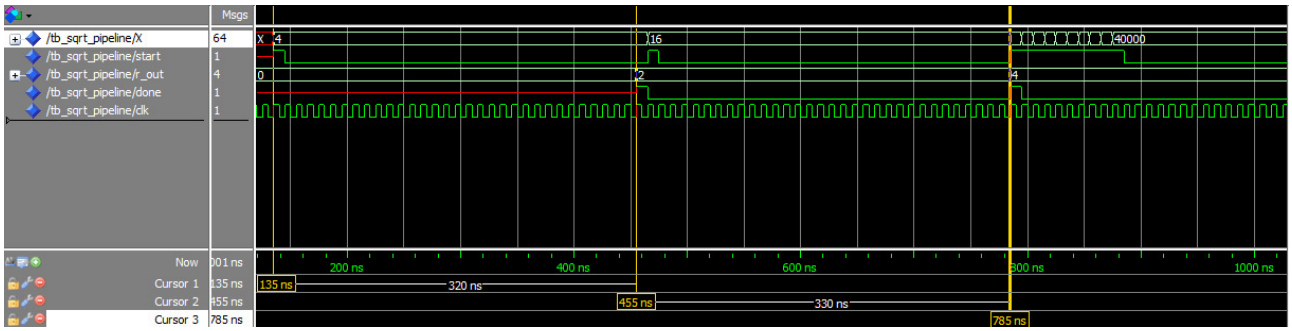


Figure 7: Diagramme temporel de pipeline

La figure 7 montre que le pipeline absorbe bien une donnée par cycle et que sa profondeur est de 32 cycles. Il important de noter qu'alimenter un tel pipeline ne sera pas évident car le Nios est incapable de charger/sauvegarder une donnée en 1 cycle. La mise est en place d'un DMA est envisageable mais ne sera vu dans ce rapport.

3 Comparaison et résultat

On remarque qu'il existe plusieurs compromis en termes de fréquences, surface FPGA. On peut par ailleurs comparer les différentes implémentations entre elles. On obtient ainsi les résultats suivants.

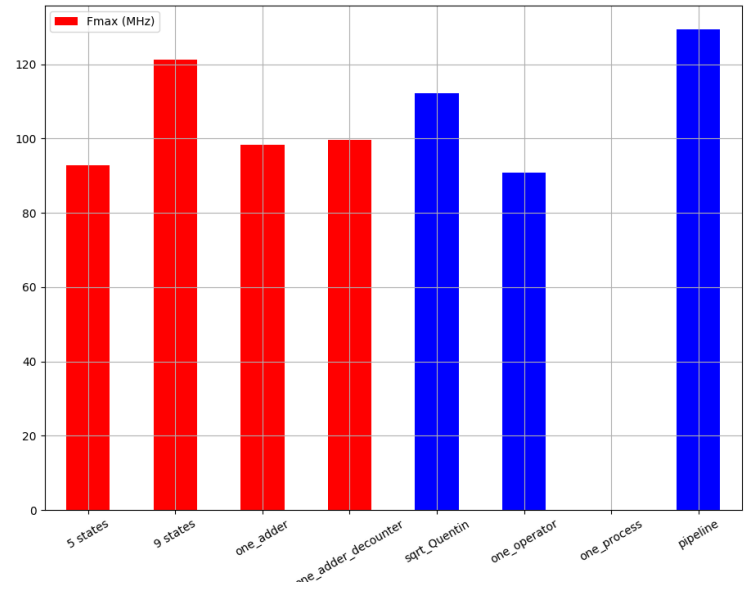


Figure 8: Mesures fréquentielles des différentes implémentations

On ne peut pas mesurer la fréquence de fonctionnement d'une circuit combinatoire sans imposer de registres à l'entrée et la sortie du circuit. On remarque que, comme attendu, on a un pipeline cadencé à une vitesse plus importante que les autres circuits. On remarque aussi que les circuits multi-cycles sont plus rapides que les circuits mono-opérateur. Ce qui est logique car elles prennent plus de place afin d'assurer une telle cadence.

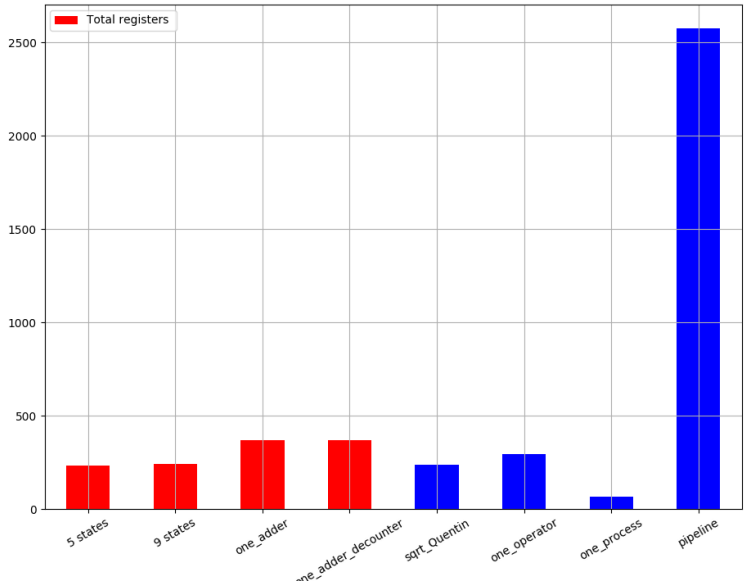


Figure 9: Usage de la surface FPGA par circuit implémenté

En terme de surface et éléments logiques occupés, la figure 9 montre que le pipeline occupé effectivement n fois plus de ressources car il y a bien n étages du même circuit. Aussi, on confirme bien les résultats trouvés en fréquentiel en observant que les mono-opérateurs sont plus économique en terme de surface FPGA.

4 Conception du système embarqué

Le système embarqué sera implémenté sur une carte Intel/Altera Cyclone II De1. Au cœur de ce système :

- un softcore le NIOS2 comme CPU
- contrôleurs mémoires :

  On-chip

  SDRAM

  SSRAM
- périphérique JTAG permettant de programmer et déboguer le NIOS
- module racine carrée VHDL

Par ailleurs, on utilisera des IP préconçues qui vont accélérer le développement du système.

4.1 Implémentation et programmation du Nios

L'IP du softcore est fourni par Intel/Altera en plusieurs versions chacune offrant différentes cadences fréquentielles, caches instructions/données, opérateurs et pipeline. On ne détaillera pas ces différences ici car elles ont été vu en cours et peuvent être retrouvées dans la documentation d'Intel. On implémente donc les différentes versions telles que :

- Nios e : cache instructions 512 o
- Nios s : cache instructions 512 o et données 512 o
- Nios s : cache instructions 2 ko et données 2 ko
- Nios f : cache instructions 512 o et données 512 o
- Nios f : cache instructions 2 ko et données 2 ko

4.2 Usage des différentes mémoires

Grâce aux outils de développement proposés par Intel/Altera, on peut semi/automatiquement instancier de systèmes embarqués avec la possibilité de paramétrer l'emplacement mémoire du programme qui sera exécuté sur CPU. On peut alors écrire un programme qui calcule la racine carrée entière d'un nombre de manière logicielle, i.e. utiliser les opérateurs du NIOS. Ensuite, placer les données et le code dans soit la mémoire On-chip, SSRAM ou SDRAM. Ceci permet d'établir un comparatif entre ces différentes mémoires en termes de conception et timings pour plusieurs version du NIOS vues en cours.

4.3 Comparaison des différentes implémentations

On exécute le même programme C sur les différentes version du Nios. On mesure à chaque fois le temps d'exécution de ce code pour une mémoire et un version Nios.

4.3.1 Mesure de temps

Il existe plusieurs manières de mesurer le temps d'exécution sur Nios. Dans notre cas, on peut soit utiliser la bibliothèque fournie par le constructeur afin d'exploiter le registre compteur du Nios. Ou bien, implémenter nous-même ces fonctions de mesures afin d'avoir un plus grand contrôle sur la mesure de temps. Sans détailler, on opte pour implémenter ces fonctions et on les compare avec la version proposée par Altera :

Implémentation	*temps de mesure à vide (μ)*
Altera	108
Personnalisée	56

On garde alors nos propres fonctions de mesures. On procède à une mesure fine du temps d'exécution. I.e. on mesure le temps de lecture depuis un tableau, d'exécution de la fonction racine carrée et de l'écriture dans un tableau qui contient 1000 éléments.

4.3.2 Temps mesurés

On obtient alors le tableau suivant :

Lecture (μs)	exécution (μs)	écriture (μs)	mémoire	Nios
951006	152117	143773	SDRAM	2k_f
920547	113891	107808	SSRAM	2k_f
864807	56543	52084	ONCHIP	2k_f
4408206	167321	174576	SDRAM	2k_s
1780954	86035	86012	SSRAM	2k_s
1332246	71018	71008	ONCHIP	2k_s
1021266	198997	142142	SDRAM	512_f
946526	124316	102657	SSRAM	512_f
911848	73247	57028	ONCHIP	512_f
4407738	167296	174586	SDRAM	512_s
1780954	86035	86012	SSRAM	512_s
1332246	71018	71008	ONCHIP	512_s
13484546	709816	713872	SDRAM	e
5308056	316000	319000	SSRAM	e
3992076	245000	248000	ONCHIP	e

Table 1: Temps de lecture, d'exécution et écriture des différentes versions pour 1000 éléments

On remarque que la version 2k_f du Nios est la plus performante. Ceci est évident car c'est elle qui exploite le plus la surface du FPGA et possède des opérateurs en pipeline plus profonds. La mémoire Onchip est plus rapide que la SRAM et encore plus rapide que la DRAM du fait de la nature de conception des ces mémoires et la rapidité des accès. On remarque aussi qu'il y a des masquages des latences du chargement par la version 2k_f.

En terme de surface FPGA, on peut voir sur le graphe suivant que la surface occupée est liée à la version du Nios. Plus la version est performante, plus elle occupera de la surface du FPGA.

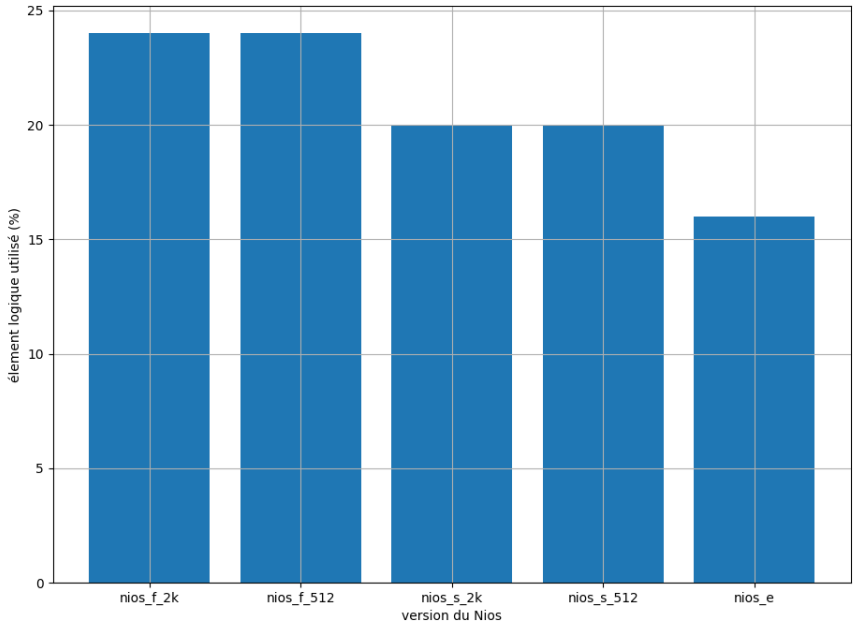


Figure 10: Occupation des éléments logiques du FPGA

On voit ici l'occupation en termes d'éléments logiques. et que les mêmes versions de Nios occupent le même nombre d'éléments logiques mais un espace mémoire plus grand pour chacun des caches.

4.4 Intégration du coprocesseur

On veut instancier l'opérateur matériel afin d'accélérer le calcul. Dans un premier temps on génère par l'outil Qsys les entrées/sorties nécessaires pour communiquer avec l'opérateur matériel.

4.4.1 Entrées/Sorties

Les ports E/S peuvent être configurés en 4 modes :

- Entrée : Le Nios ne peut que lire cette adresse mémoire
- Sortie : Le Nios ne peut qu'écrire dans cette adresse mémoire
- Entrée/Sortie : Le Nios peut lire et écrire au choix
- E/S mixte : Le Nios peut lire et écrire au choix

Ces implémentations sont vues différemment au point de vu du VHDL. L'E/S mixte est un signal vu comme un *inout*. Alors que le Entrée/Sortie est vu en 2 signaux mais vu comme une seule variable coté C.

Au final, on a besoin des ports suivants :

- Start/Done : 1 bit en entrée/sortie
- X : 32 bits sortie

- Z : 32 bits entrée

Il suffit alors de créer une fonction qui positionne les variables puis lance les calculs en mettant *Start* à 1 et attend (ou non) que *Done* soit à 1. Le *reset* est connecté au reset Nios. Une possible implémentation de la fonction est donnée ainsi :

```
1  ushort sqrt_hw (uint X)
2  {
3          ushort z=0;
4          *SQRT_X=X;
5          *SQRT_status=1;
6          while(0x1 & *SQRT_status != 1);
7          *SQRT_status=0;
8          z=*SQRT_result;
9          return z;
10 }
```

On peut choisir ou non d'enlever la ligne 7 afin d'avoir un code non-bloquant car le CPU est totalement libre durant tout le calcul.

4.5 Intégration des instructions personnalisées

On souhaite alors comparer ces implémentations avec une dernière qui va virtuellement augmenter l'ALU du Nios en ajoutant un des opérateurs racine carrée VHDL à ses opérateurs. Cette procédure consiste simplement à indiquer à l'outil Qsys quel circuit mettre comme opérateur ensuite le synthétiser afin d'avoir une macro-instruction qui agira comme une fonction. Un résultat possible de cette méthode : `ALT_CI_SQRT_MODULE_0(X)`

4.6 Comparaison

On compare les temps d'exécution des 2 implémentations précédentes au meilleur temps d'exécution (+lecture + écriture) du tableau 1 pour la version `2k_f`. On obtient les résultats suivants :

Implémentation	Temps de mesuré (μs) pour 1000 éléments
Logicielle	973434
Coprocesseur	297633
Instruction personnalisée	149633

Table 2: Comparatif entre les différentes implémentations

On remarque que les implémentations matérielles sont beaucoup plus rapides que les implémentations logicielles. Ceci est naturel car c'est une des propriétés des conceptions en flot de données par rapport à une conception logicielle. On voit aussi que l'implémentation Coprocesseur est plus lente que l'instruction personnalisée car elle utilise de ports IO asynchrones pour communiquer avec le modules SQRT. On peut dire que l'instruction personnalisée est plus proche du CPU et donc plus rapide à interfacer. Par contre elle occupe le CPU pendant cet usage. Là où l'instruction personnalisée peut agir d'une manière non bloquante (type fire & forget).

5 Conclusion

Durant toutes ces séances de TP, on aura vu plusieurs aspects étudiés lors de ces dernières années. Que ce soit l'implémentation de brique VHDL ou les différents mécanismes dans une architectures CPU, les outils Intel/Altera facilement la mise en place de ces systèmes toujours plus compliqués.

Dans un premier temps, on a étudié et testé les différentes implémentations VHDL de l'algorithme racine carrée. On a pu les comparer en terme de latence et de surface FPGA. Dans un second temps, on a conçu un système électronique entièrement basé sur une carte FPGA. Ce système est facilement paramétrable d'un point de vu haut niveau même s'il repose sur plusieurs construction bas niveau qui nous sont abstraites.

Par la suite, une fois le système implémenté, on a joué sur les différents curseurs à disposition pour pouvoir comparer l'exécution des différentes implémentations logicielles en utilisant les différentes mémoires et versions du Nios. On a pu voir l'effet des outils de mesures du temps sur ces performances.

Enfin on a comparé ces implémentations logicielles au implémentations hybrides utilisant une brique VHDL. Le constat est sans appel, les implémentations hybrides sont toujours plus rapides quelque soit l'interfaçage. Par manque de temps, on n'a pas pu implémenter l'usage du bus Avalon. Ceci n'aurait que confirmer ce résultat.