Département des Technologies de l'information
et de la communication (TIC)

Informatique et systèmes de communication

System-On-Chip FPGA

SCF

Laboratoire 8

Cordic

Etudiant André Costa, Lucas Lattion & Patrick Maillard
Professeur Alberto Dassatti & Yann Thoma
Assistant Anthony I. Jaccard
Année 2025

Yverdon-les-Bains, 13.05.2025

Table des matières

1. Introduction

Le traitement numérique du signal requiert fréquemment des transformations entre différentes représentations de données, notamment entre les coordonnées cartésiennes et polaires. Dans les systèmes numériques embarqués, où les ressources de calcul sont souvent limitées, l'implémentation efficace de fonctions trigonométriques devient un défi majeur. L'algorithme CORDIC (COordinate Rotation DIgital Computer) offre une solution élégante à ce problème en permettant d'effectuer ces calculs complexes sans recourir à des multiplicateurs dédiés ou à des tables de correspondance volumineuses.

Ce rapport présente la conception, l'implémentation et la vérification d'un calculateur CORDIC fonctionnant en mode "vectoring" pour transformer des coordonnées cartésiennes (re, im) en coordonnées polaires (amplitude, phase). Trois architectures distinctes ont été explorées : combinatoire, pipeline et séquentielle, chacune offrant un compromis différent entre latence, débit, fréquence maximale de fonctionnement et utilisation des ressources matérielles.

2. Guide de démarrage rapide

Voici comment lancer les différents tests:

```
mkdir -p code/sim
cd code/sim

# Lancer les tests pour chaque architecture
# NOTE: Pressez sur NO quand le test actuel fini pour lancer la prochaine
architecture
vsim -do "do ../scripts/sim.do all"

# Architecture combinatoire (par défaut)
vsim -do "do ../scripts/sim.do comb 0"

# Architecture pipeline
vsim -do "do ../scripts/sim.do pipeline 1"

# Architecture séquentielle
vsim -do "do ../scripts/sim.do sequential 2"

# Bloc de pré-traitement
vsim -do "do ../scripts/sim_bloc.do pre_test"

# Bloc de cordic itération
vsim -do "do ../scripts/sim_bloc.do cordic_iteration_test"

# Bloc de post-traitement
vsim -do "do ../scripts/sim_bloc.do post_test"
```

3. Implémentation du calculateur CORDIC

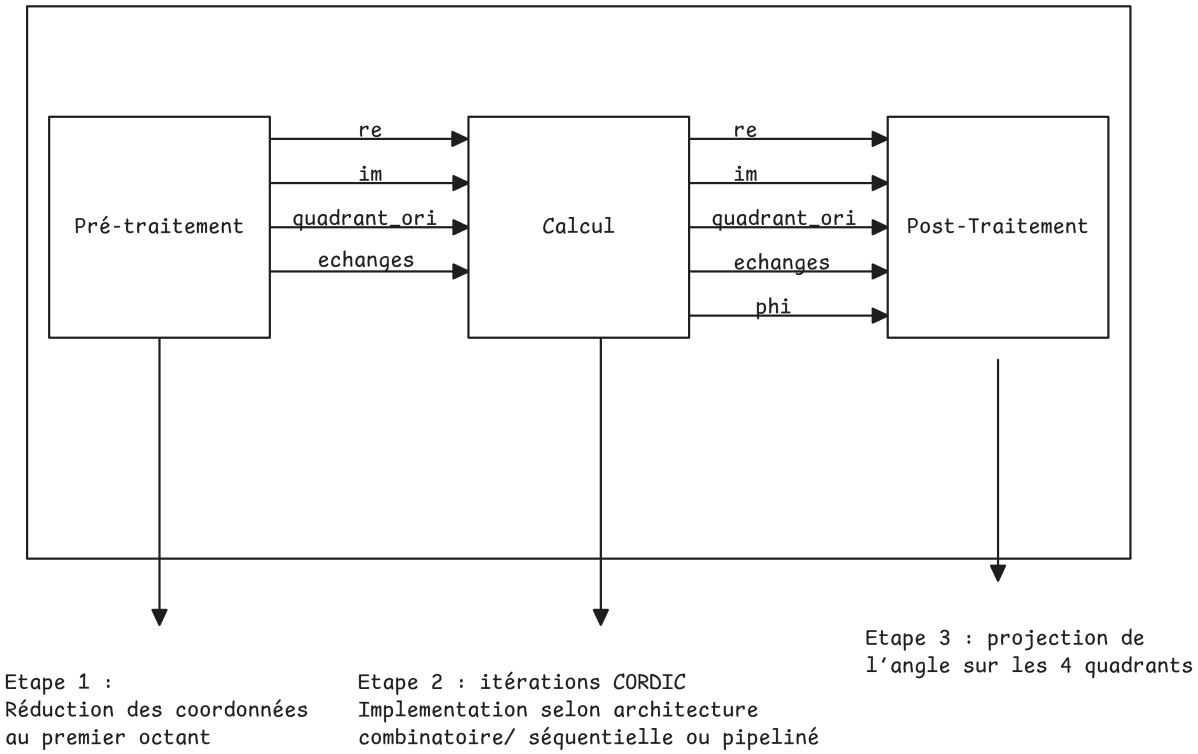
3.1. Architecture générale

L'implémentation du calculateur CORDIC a été décomposée en trois composants principaux afin de faciliter la réutilisation entre les différentes architectures et d'améliorer la lisibilité du code :

- **Composant de prétraitement** (`cordic_pre_treatment`) : Réalise la première étape de l'algorithme CORDIC en projetant les coordonnées dans le premier octant.
- **Composant d'itération** (`cordic_iteration`) : Effectue une itération de l'algorithme CORDIC selon les formules définies dans le cahier des charges.
- **Composant de post-traitement** (`cordic_post_treatment`) : Applique les corrections nécessaires pour obtenir la phase correcte en fonction du quadrant d'origine et extrait l'amplitude du vecteur.

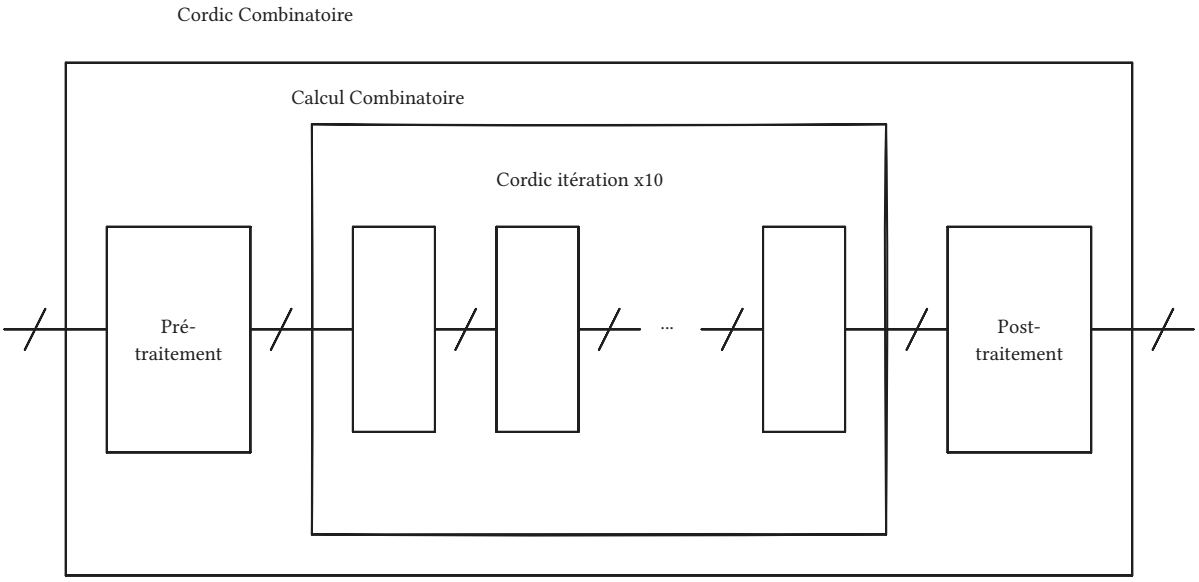
Cette décomposition modulaire permet d'implémenter les trois architectures demandées (combinatoire, pipeline, séquentielle) en réutilisant ces blocs de base et en modifiant uniquement la manière dont ils sont connectés et synchronisés.

Schéma bloc



3.2. Architecture combinatoire

3.2.1. Schéma bloc



3.2.2. Principe de fonctionnement

L'architecture combinatoire instancie les composants de manière purement combinatoire, sans registres entre les étapes de calcul. Elle est caractérisée par :

- Un bloc de prétraitement
- 10 blocs d'itération CORDIC connectés en série
- Un bloc de post-traitement

3.2.3. Caractéristiques

- **Avantages** : Latence minimale (le résultat est disponible immédiatement après l'application des entrées).
- **Inconvénients** : Chemin critique très long traversant tous les composants, limitant significativement la fréquence maximale de fonctionnement.

3.2.4. Fréquence Maximale

Pour cette architecture, il n'est pas possible de calculer la fréquence maximale, dû au fait qu'il n'y a pas de registres pour dans le système synthétisé. Cependant nous pouvons en déduire de la fréquence maximale de l'architecture pipeliné (voir ci-dessus).

L'architecture pipeliné tourne à `304,4MHz` ce qui implique un temps de propagation maximale de:

$$\frac{1}{304 * 10^6} = 3\text{ns}$$

Si l'on considère que chaque bloc est équivalent, nous pouvons en déduire que le temps de propagation de notre système sera de

$$12 * \frac{1}{304 * 10^6} = 39\text{ns}$$

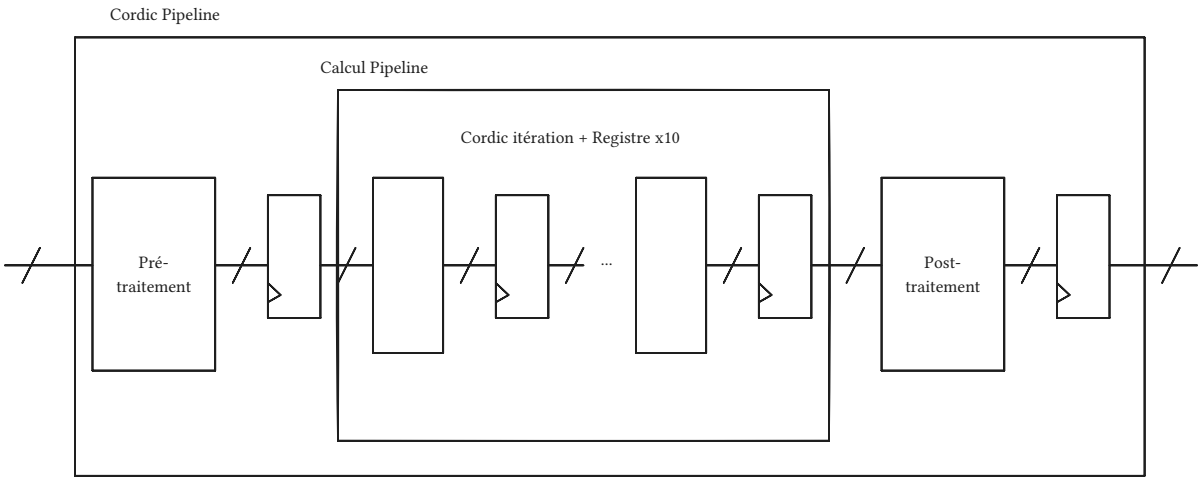
et donc, notre bloc limiterait la fréquence maximale dans un système plus complet à

$$\frac{1}{12 * \frac{1}{304*10^6}} = 25.6 \text{ MHz}$$

Note: Le 12 dans notre calcul prend en compte les 10 blocs itération et les 2 blocs pré et post traitement.

3.3. Architecture pipeline

3.3.1. Schéma bloc



3.3.2. Principe de fonctionnement

L'architecture pipeline insère des registres entre chaque étape de calcul pour découper le chemin critique en segments plus courts. Elle comprend :

- Un bloc de prétraitement suivi d'un registre
- 10 blocs d'itération CORDIC, chacun suivi d'un registre
- Un bloc de post-traitement
- Une gestion du flux de données avec des signaux de contrôle permettant de stopper le pipeline si nécessaire

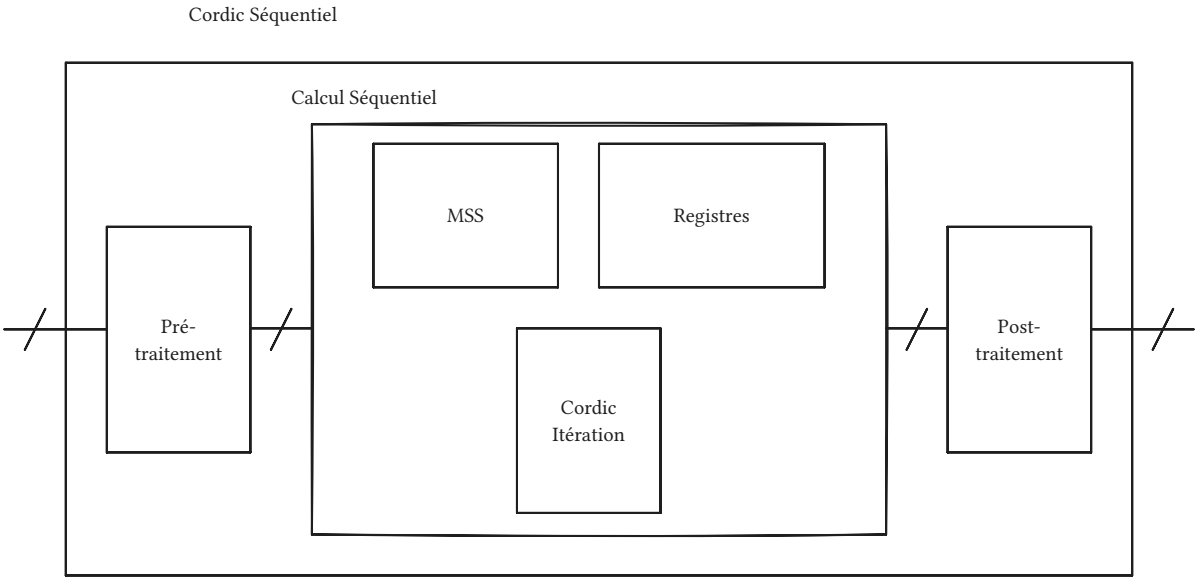
Le schéma bloc de cette architecture est très similaire à l'architecture combinatoire. En effet, nous ajoutons des registres entre chaque étape pour garantir le plus petit temps de propagation possible. La partie la plus compliqué est de gérer correctement l'arrêt de la pipeline, pour cela, notre implémentation permet aux données d'arriver à la dernière étape avant de s'arrêter au cas où le composant qui vient après n'est pas prêt.

3.3.3. Caractéristiques

- **Avantages** : Fréquence de fonctionnement élevée, débit maximal (un résultat par cycle d'horloge en régime permanent).
- **Inconvénients** : Latence importante (12 cycles entre l'entrée et la sortie), consommation de ressources plus élevée pour les registres.
- **Fréquence maximale** : 304.4 MHz - Compilation avec Quartus pour la carte DE1-SoC - modèle Slow 1100mV 85C

3.4. Architecture séquentielle

3.4.1. Schéma bloc



3.4.2. Principe de fonctionnement

L'architecture séquentielle utilise une machine à états finis pour contrôler la réutilisation d'un seul bloc d'itération. Elle est composée de :

- Un bloc de prétraitement
- Un seul bloc d'itération CORDIC utilisé séquentiellement 10 fois
- Un bloc de post-traitement
- Une machine à états finis qui gère le séquencement des étapes

Les états principaux de la machine à états sont :

- `IDLE` : Attente d'une nouvelle entrée
- `ITERATION` : Exécution séquentielle des 10 itérations CORDIC
- `POST_TREATMENT` : Application du post-traitement
- `VALID` : Signalement que le résultat est valide

Même pour cette architecture, nous pouvons utiliser les mêmes blocs utilisées auparavant. Ceci montre l'élégance de la décomposition effectué au début de l'implémentation.

3.4.3. Caractéristiques

- **Avantages** : Utilisation minimale de ressources, particulièrement adapté pour les applications où la surface est critique.
- **Inconvénients** : Débit limité (un résultat tous les 12 cycles d'horloge), latence moyenne (12 cycles).
- **Fréquence maximale** : 195.85 MHz - Compilation avec Quartus pour la carte DE1-SoC - modèle Slow 1100mV 85C

3.5. Comparaison des performances

Le tableau suivant présente une comparaison des performances des trois architectures implémentées :

Architecture	Fréquence maximale	Latence (cycles)	Débit max (résultats/cycle)	Utilisation de ressources
Combinatoire	25.2 MHz	1	1	Faible
Pipeline	304.4 MHz	12	1	Élevée
Séquentielle	195.85 MHz	12	1/12	Minimale

3.6. Conclusion implémentation

Les trois architectures implémentées pour le calculateur CORDIC offrent différents compromis entre fréquence de fonctionnement, latence, débit et utilisation des ressources. Le choix de l'architecture dépend donc des contraintes spécifiques de l'application :

- L'architecture **combinatoire** est adaptée aux applications nécessitant une latence minimale sans contrainte forte sur la fréquence.
- L'architecture **pipeline** convient parfaitement aux applications à haut débit requérant une fréquence de fonctionnement élevée.
- L'architecture **séquentielle** est idéale pour les applications où les ressources matérielles sont limitées et où le débit n'est pas critique.

La modularité de notre implémentation facilite le passage d'une architecture à l'autre, ce qui permet d'adapter facilement le calculateur CORDIC aux besoins spécifiques de différentes applications.

4. Calcul de référence

Lors de l'implémentation de l'algorithme, nous nous sommes retrouvées avec de gros décalages entre la valeur théorique et la valeur calculé par le système. La valeur attendue du système a donc été calculé à la main pour vérifier que le système répond à la donnée.

$$\text{re} = 1000 \ \text{im} = 500$$

4.1. Étape 1 : Prétraitement

- Calcul de la valeur absolue de re et im. Ceci projette les coordonnées dans le premier quadrant.
- Comparaison entre re et im. Si im > re alors leurs valeurs sont échangées. Ceci projette les coordonnées dans le premier octant.
  - Coordonnées initiales :

    $$\text{re} = 1000 \ \text{im} = 500$$
  - Les deux sont positifs : Premier quadrant
  - Valeurs absolues : Pas de changement car déjà positifs
  - re > im : Pas d'échange

4.2. Étape 2 : Itérations CORDIC

Les constantes alpha_const sont fournies dans le fichier `cordic_pkg.vhd` :

- α_1 = 302 (= 00100101110 en binaire)
- α_2 = 160 (= 00010100000)
- α_3 = 81 (= 00001010001)
- α_4 = 41 (= 00000101001)
- α_5 = 20 (= 00000010100)
- α_6 = 10 (= 00000001010)
- α_7 = 5 (= 00000000101)
- α_8 = 3 (= 00000000011)
- α_9 = 1 (= 00000000001)
- α_{10} = 1 (= 00000000001)

4.2.1. Explication sur les shifts arithmétiques vs division entière

Dans l'implémentation matérielle du CORDIC, les divisions par puissances de 2 sont réalisées par des décalages binaires (shifts). Il est important de comprendre la différence entre une division entière classique et un décalage arithmétique, particulièrement pour les nombres négatifs :

- **Division entière** : Lorsqu'on calcule manuellement, on obtient par exemple −156 / 16 = −9,75, qui est tronqué à −9 (arrondi vers zéro)
- **Décalage arithmétique à droite (-156 >> 4)** : Le décalage préserve le bit de signe et arrondit vers le bas (vers -∞), donnant −10 en complément à deux

Cette différence peut expliquer les écarts entre un calcul manuel théorique en utilisant la division entière et l'implémentation matérielle réelle. Afin de vérifier notre implémentation matérielle, nous avons vérifié les calculs à l'aide de shift.

4.2.2. Calculs des itérations

À chaque itération, les calculs suivants sont effectués selon le signe de `im` :

- Si la partie imaginaire à l'itération i est négative :
  - $\text{re}_{\{i+1\}} = \text{re}_i - \frac{\text{im}_i}{2^i}$ (où la division est un décalage arithmétique)
  - $\text{im}_{\{i+1\}} = \text{im}_i + \frac{\text{re}_i}{2^i}$
  - $\varphi_{\{i+1\}} = \varphi_i - \text{alpha\_const}_i$

- Si la partie imaginaire à l'itération i est positive :
  - $\text{re}_{\{i+1\}} = \text{re}_i + \frac{\text{im}_i}{2^i}$
  - $\text{im}_{\{i+1\}} = \text{im}_i - \frac{\text{re}_i}{2^i}$
  - $\varphi_{\{i+1\}} = \varphi_i + \text{alpha\_const}_i$

Le tableau jusqu'à 10 itérations (avec décalages arithmétiques) :

Iter	re_i	im_i	phi_i	im < 0 ?	im_i>>i	re_i>>i	re_i+1	im_i+1	phi_i+1
Init	1000	500	0	-	-	-	-	-	-
1	1000	500	0	NON	250	500	1250	0	302
2	1250	0	302	NON	0	312	1250	−312	462
3	1250	−312	462	OUI	−39	156	1289	−156	381
4	1289	−156	381	OUI	−10	80	1299	−76	340
5	1299	−76	340	OUI	−3	40	1302	−36	320
6	1302	−36	320	OUI	−1	20	1303	−16	310
7	1303	−16	310	OUI	−1	10	1304	−6	305
8	1304	−6	305	OUI	−1	5	1305	−1	302
9	1305	−1	302	OUI	−1	2	1306	1	301
10	1306	1	301	NON	0	1	1306	0	302

Détails des calculs (avec décalages arithmétiques pour les nombres négatifs) :
- Itération 1 : $\text{im}_i \gg 1$ = 500>>1 = 250, $\text{re}_i \gg 1$ = 1000>>1 = 500
- Itération 2 : $\text{im}_i \gg 2$ = 0>>2 = 0, $\text{re}_i \gg 2$ = 1250>>2 = 312
- Itération 3 : $\text{im}_i \gg 3$ = (-312)>>3 = −39, $\text{re}_i \gg 3$ = 1250>>3 = 156
- Itération 4 : $\text{im}_i \gg 4$ = (-156)>>4 = −10, $\text{re}_i \gg 4$ = 1289>>4 = 80
- Itération 5 : $\text{im}_i \gg 5$ = (-76)>>5 = −3, $\text{re}_i \gg 5$ = 1299>>5 = 40
- Itération 6 : $\text{im}_i \gg 6$ = (-36)>>6 = −1, $\text{re}_i \gg 6$ = 1302>>6 = 20
- Itération 7 : $\text{im}_i \gg 7$ = (-16)>>7 = −1, $\text{re}_i \gg 7$ = 1303>>7 = 10
- Itération 8 : $\text{im}_i \gg 8$ = (-6)>>8 = −1, $\text{re}_i \gg 8$ = 1304>>8 = 5
- Itération 9 : $\text{im}_i \gg 9$ = (-1)>>9 = −1, $\text{re}_i \gg 9$ = 1305>>9 = 2
- Itération 10 : $\text{im}_i \gg 10$ = 1>>10 = 0, $\text{re}_i \gg 10$ = 1306>>10 = 1

Résultat de l'étape 2 :

$$\text{re} = 1306; \text{im} = 0; \varphi = 302$$

4.3. Étape 3 : Projection de l'angle sur les 4 quadrants

4.3.1. Projection sur le premier quadrant :

On laisse `phi` tel quel vu que les valeurs `re` et `im` non pas été modifiées lors de l'étape 1.

4.3.2. Projection sur les quatre quadrants :

On laisse encore `phi` tel quel car le quadrant d'origine était le premier.

Résultat de l'étape 3 :

$$\text{re} = 1306; \text{im} = 0; \varphi = 302$$

4.4. Étape 4 : Extraction de l'amplitude

L'algorithme CORDIC en mode "vectoring" rabat le vecteur sur l'axe des réels. L'amplitude est donc simplement la valeur réelle de la dernière itération.

Résultat de l'étape 4 :

$$\text{amp} = 1306; \varphi = 302$$

4.5. Résultats finaux du calculateur CORDIC :

4.5.1. Conversion phase en radians

Conversion de la phase en radians :

- 302 sur 11 bits signés correspond à : $\frac{302}{2^{10}} * \pi \approx 0.295 * \pi \approx 0.926$ radians

4.5.2. Calcul Théorique

- Amplitude théorique:

$$\sqrt{1000^2 + 500^2} \approx 1118$$

- Phase théorique

$$\arctan\left(\frac{500}{1000}\right) \approx 0.464 \text{ radians}$$

4.5.3. Comparaison

	Théorique	**Algorithme**
Amplitude	1118	1306
Phase	0.464	0.926

4.6. Conclusion calcul de référence

Ces résultats montrent que l'approximation réalisée par cet algorithme n'est pas suffisamment précise. Par conséquent, il n'est pas pertinent de se baser sur la valeur théorique lors de la vérification dans le test bench.

Pour pallier cela, nous avons utilisé le même algorithme dans le test bench que celui implémenté dans notre système. Cette approche permet de s'assurer que l'implémentation fonctionne correctement. En temps normal, cette méthode ne serait pas recommandée, car elle ne permet pas de valider la justesse de l'algorithme lui-même. Toutefois, étant donné les performances limitées de cet algorithme en termes de précision, cela reste la seule solution fiable pour évaluer l'exactitude de notre implémentation.

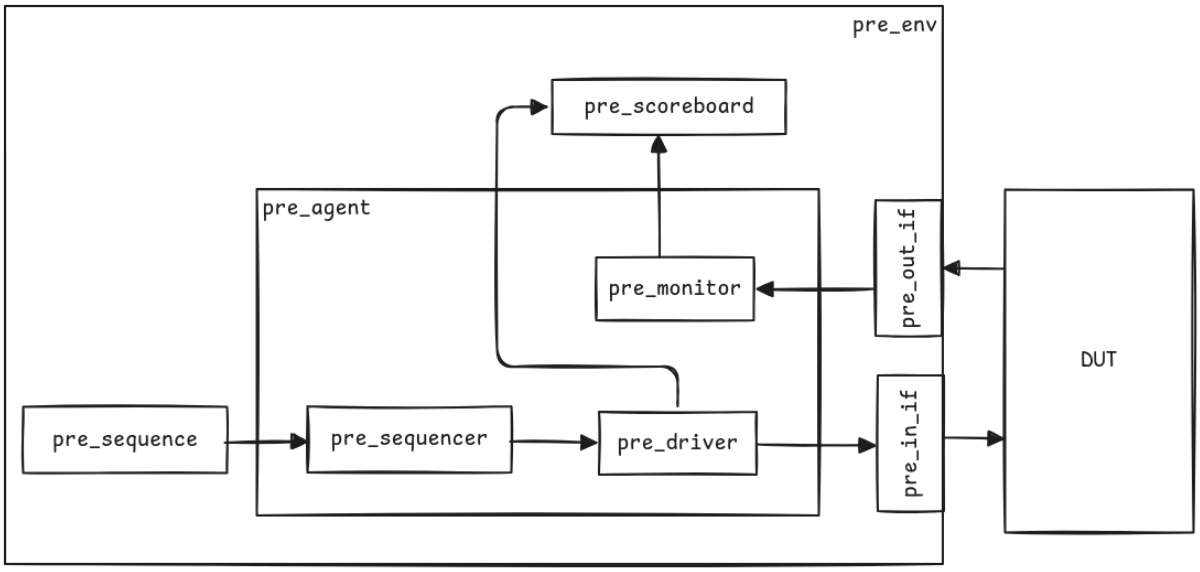
5. CORDIC Vérification - UVM

5.1. introduction

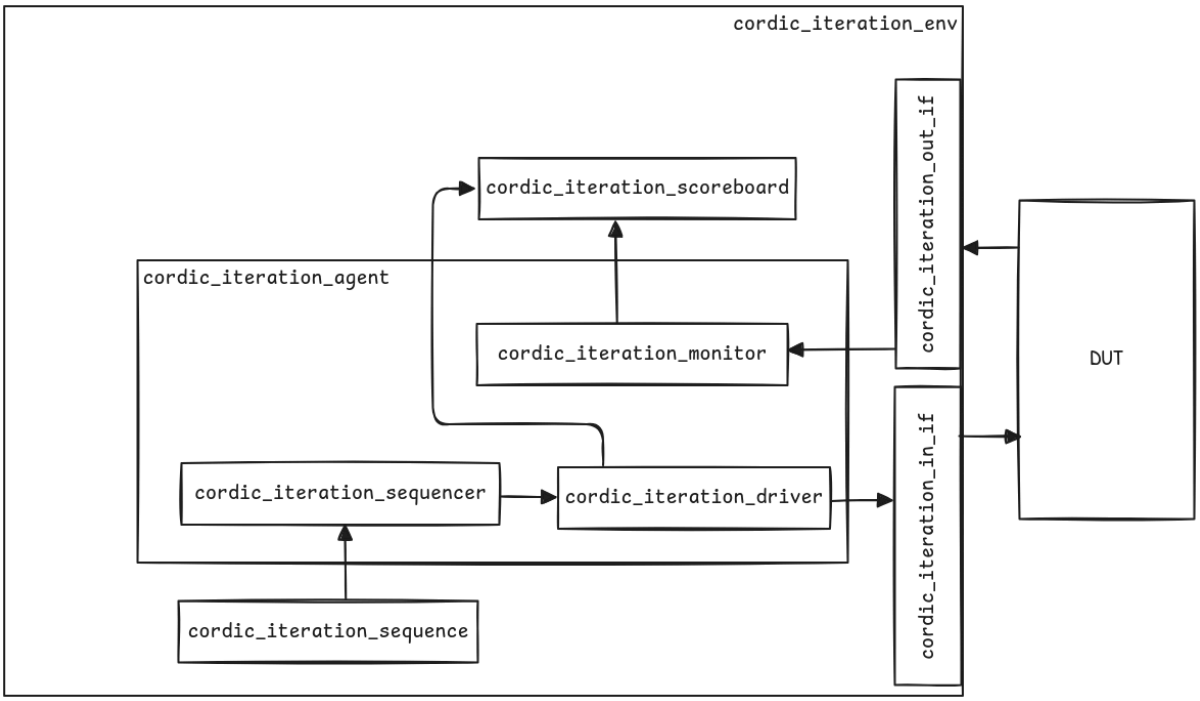
La conception du testbench de ce laboratoire devait normalement se faire de manière simple, c'est à dire que l'on devait tout simplement concevoir un module simple testant tous les cas possible du CORDIC avec tout simplement 4 entrée et 4 sorties, mais suite à la conception du système nous nous somme retrouvés avec 3 bloc qui complétaient le module complet de CORDIC, un bloc de pré-traitement, un bloc de calcul pour CORDIC et un bloc de post-traitement. De ce fait afin de se familiariser un peu avec la réalisation d'un testbench réel, un UVM a été réalisé en s'inspirant de la documentation officiel de ce lien.

5.2. Schéma du testbench

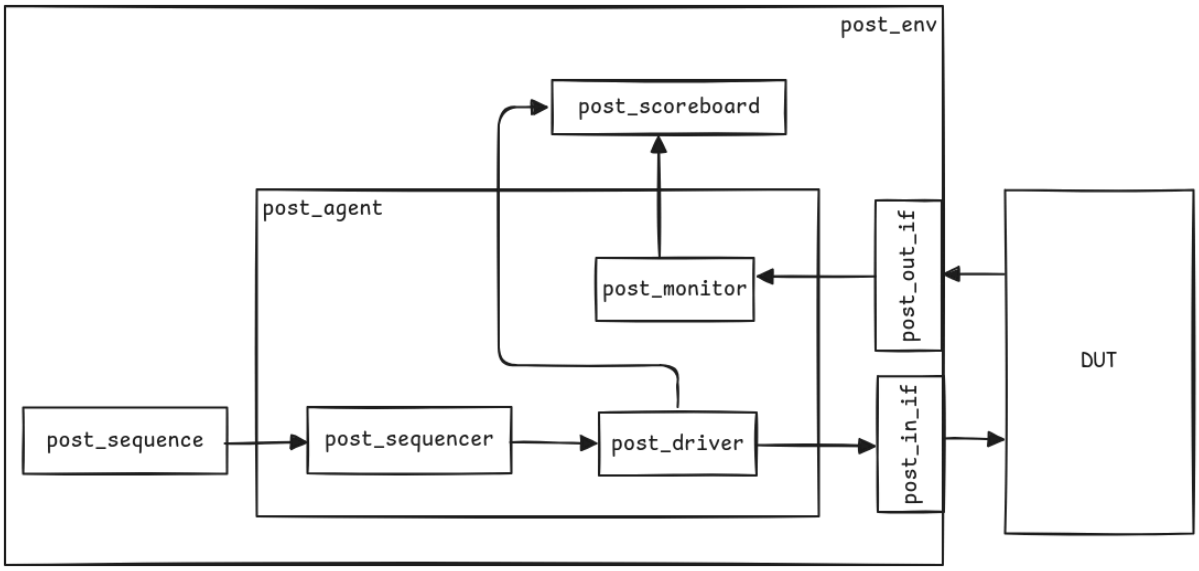
5.2.1. Pré-traitement



5.2.2. Cordic-itération-traitement

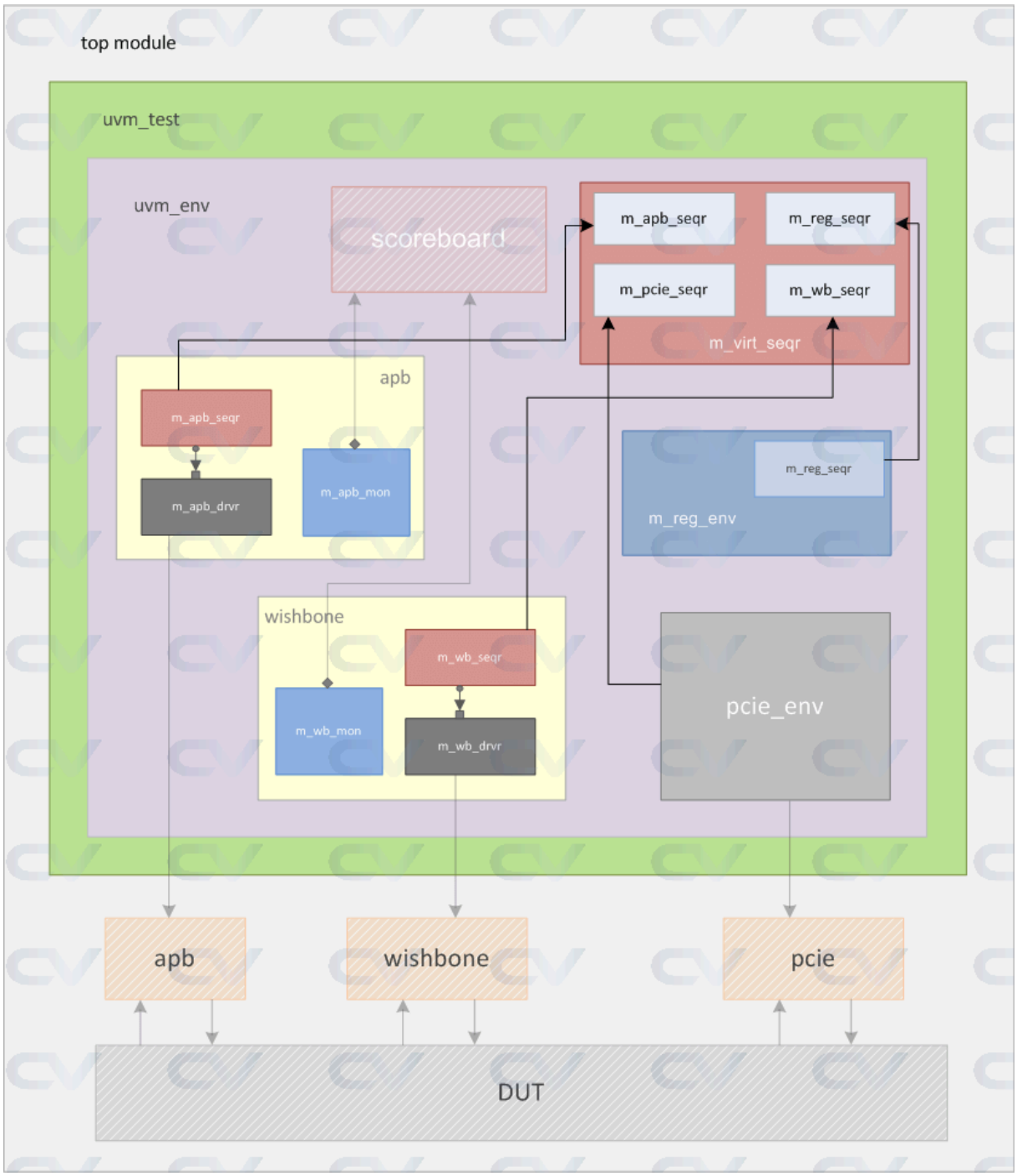


5.2.3. Post-traitement



5.3. Choix de conception

Théoriquement afin d'avoir un UVM complet il aurais fallu regrouper chacun des environnements dans un seul grand environnement afin de tester le système complet, cela aurait permis de récupérer les valeurs de chacun des bloc durant tout le processus. Cela utiliserais le même princpipe que le schéma ci-dessous:



Malheureusement n'ayant pas le temps de le réaliser et comprendre ce type de système, la partie vérifiant le système complet à été réaliser dans un autres fichier à l'extérieur de l'UVM, n'utilisant aucun des concept de ce dernier.

5.4. Stimulus et contrainte

Afin de réaliser 100% des cas possible sur les différents bloc chacun des bloc se retrouve avec son coverage ainsi que ces contraintes, cela nous permet d'être certain du bon fonctionnement de chaque bloc.

5.4.1. Pré-traitement

La fonctionnalité principale du pré-traitement est tout simplement de ramener les cordonnée cartesienne vers le premier octant (total de 8 octants), donc afin de vérifier cela le sequencer génère

une centaine de valeur aléatoire ce qui garanti de tester la majorité des cas. Afin de savoir si tout les octant on bien été couvers une variable nommé octant à été ajouté afin de vérifier à l'aide d'un coverage que chacun des octants a été testé.

Contraintes:

Afin d'assurer le bon fonctionnement de la randomisation voici les contrainte utilisée:
- range: défini les valeurs min et max que peuvent prendre re et im
- my_octant: défini quels valeurs appartient à quel octant

Coverage:

Le coverage all_octant vérifie tout simplement que tout les octants ont été testé.

5.4.2. Cordic-itération-traitement

Pour cette partie l'objectif principal est de vérifier que le calcul est correct pour chaque itération possible, donc entre 1 et 10.

Contraintes:

voici les contraintes utilisée:
- range: défini les valeurs min et max que peuvent prendre re, im, phi et iter
- re_bigger_im: s'il s'agit de la première itération alors re dois être plus grand ou égal à im

Coverage:

Le coverage cordic_iteration_cg vérifie tout simplement que toutes les itérations ont été testé.

5.4.3. Post-traitement

Afin d'assurer un bon fonctionnement de ce bloc il nous faut vérifier que pour chaque quadrant et chaque changement de signal que le calcul soit fait correctement.

Contraintes:

Voici donc la contrainte utilisée:
- range: défini les valeurs min et max que peuvent prendre re, im et phi

Coverages:

Le coverage post_cg vérifie que chaque quadrant ont été testé avec des changement de signal ou non (cross).

5.5. Scoreboard

Chacun des scoreboards reçois les transactions du driver et du moniteur au travers d'une FIFO, voici à quoi ressemble les envoies au scoreboard:

monitor/driver ⟶ analysis_port ⟶ FIFO ⟶ scoreboard

une fois les valeurs reçu le scoreboard calcule en premier lieu le résultat à obtenir avec les valeurs d'entrée puis les compare aux valeur reçu du DUT.

6. Conclusion

Ce projet de calculateur CORDIC a permis d'explorer différentes stratégies d'implémentation matérielle d'un algorithme mathématique complexe sur FPGA. L'approche modulaire adoptée, divisant le traitement en trois étapes distinctes (prétraitement, itérations CORDIC et post-traitement), s'est révélée particulièrement efficace pour faciliter le développement et la réutilisation du code à travers les trois architectures proposées.

Les résultats obtenus mettent en évidence les compromis inhérents à chaque architecture :

- L'architecture combinatoire offre une latence minimale (1 cycle) mais une fréquence de fonctionnement limitée à environ 25,2 MHz en raison de son long chemin critique.
- L'architecture pipeline atteint la fréquence la plus élevée (304,4 MHz) avec un débit maximal d'un résultat par cycle, au prix d'une latence de 12 cycles et d'une consommation de ressources plus importante.
- L'architecture séquentielle présente une utilisation minimale de ressources tout en maintenant une fréquence respectable de 195,85 MHz, mais son débit est limité à un résultat tous les 12 cycles.

La vérification du système a été réalisée à l'aide d'une méthodologie UVM (Universal Verification Methodology), permettant de tester indépendamment chaque composant et d'atteindre une couverture de test de 100% pour toutes les fonctionnalités. Cette approche rigoureuse a confirmé le bon fonctionnement du système dans toutes les configurations possibles. Nous avons également noté une divergence entre les résultats théoriques et ceux calculés par notre implémentation. Cette différence s'explique par les approximations inhérentes à l'algorithme CORDIC, notamment lors des divisions par décalage arithmétique. Cette observation souligne l'importance d'adapter les méthodes de vérification à la nature de l'algorithme plutôt que de se fier uniquement aux valeurs théoriques idéales.

En conclusion, ce projet illustre la puissance de l'algorithme CORDIC pour implémenter efficacement des fonctions mathématiques complexes sur FPGA, tout en démontrant l'impact des choix architecturaux sur les performances du système. La méthodologie de conception et de vérification employée offre un cadre robuste pour le développement de systèmes numériques complexes, adaptable à diverses contraintes de conception.