Compte Rendu

TP-Algorithme du « Cordic »

M1. Master en Systèmes Embarqués et Systèmes Intégrés

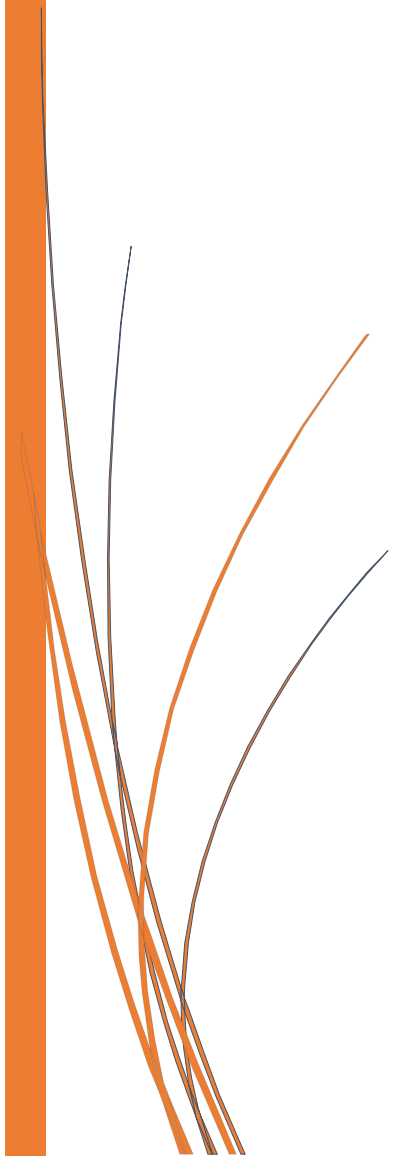
Projet Cordic - Master 1

Étudiant : JAVIER LÓPEZ José Antonio

No. Etudiant : 21804920

Date de livraison :06 décembre 2018

Professeur : TANGUY Philippe



Faculté sciences & sciences de l'ingénieur

Université Bretagne Sud

Sommaire

1. Introduction

1.1. Objectif du TP

L'objectif de ce TP sera de réaliser la conception et l'implémentation de l'algorithme Cordic en utilisant un langage de description matériel destiné à représenter le comportement ainsi que l'architecture d'un système électronique numérique (VHDL).

1.2. Rappel théorique

CORDIC est l'acronyme de COordinate Rotation DIgital Computer, qui en français signifie Ordinateur Numérique pour la Rotation de coordonnées. L'algorithme original a été proposé par Jack Volder en 1959. La rotation vectorielle peut être utilisée pour convertir des systèmes de coordonnées (cartésiennes en polaires et vice versa), l'amplitude des vecteurs et dans le cadre de fonctions mathématiques plus complexes telles que la transformation de Fourier rapide (Fast Fourier Transform, TFR) et la transformation en cosinus discret (Discrete Cosine Transform, TCD) (Wikipedia, 2018).

L'algorithme CORDIC fournit une méthode itérative pour faire tourner les vecteurs à certains angles et est basé exclusivement sur les sommes et les déplacements. L'algorithme original tel que proposé initialement décrit la rotation d'un vecteur bidimensionnel dans le plan cartésien. L'algorithme CORDIC fonctionne normalement selon deux modes. La première est appelée rotation, qui fait tourner le vecteur mis en paramètre à un angle spécifique. Le second mode, appelé vectorisation, fait tourner le vecteur d'entrée sur l'axe X, accumulant ainsi l'angle nécessaire pour effectuer cette rotation. Dans le cas d'une rotation, l'accumulateur angulaire est initialisé avec l'angle à faire tourner. La décision sur le sens de rotation à chaque étape d'itération est prise pour minimiser l'amplitude de l'angle accumulé. Par conséquent, le signe qui détermine le sens de rotation est obtenu à partir de la valeur de cet angle à chaque pas.

Pour une vectorisation, l'angle introduit est tourné pour l'aligner avec l'axe X. Pour obtenir ce résultat, au lieu de minimiser l'amplitude de l'accumulateur angulaire, on minimise l'amplitude de la composante y, puisque si y = 0 alors le vecteur est sur l'axe X. On utilise également le signe du composant et pour déterminer le sens de rotation. Si l'accumulateur angulaire est initialisé avec zéro, à la fin du processus, il contiendra l'angle de rotation approprié (Brunelle, 2015) .

Les équations pour le mode de rotation et le mode vectorisation sont les suivantes :

Tableau 1. Formules du Cordic

Rotation	Vectorisation
$X_{i+1} = X_i - Y_i d_i 2^{-i}$ $Y_{i+1} = Y_i + X_i d_i 2^{-i}$	$X_{i+1} = X_i - Y_i d_i 2^{-i}$ $Y_{i+1} = Y_i + X_i d_i 2^{-i}$

$Z_{i+1} = Z_i - d_i\varepsilon_i$ Où $d_i \begin{cases} -1\,, & si\ Z_i < 0 \\ 1\,, & si\ Z_i \geq 0 \end{cases}$	$Z_{i+1} = Z_i - d_i\varepsilon_i$ Où $d_i \begin{cases} -1\,, & si\ Y_i \geq 0 \\ 1\,, & si\ Y_i < 0 \end{cases}$
Où $\varepsilon_i = arctg(2^{-i})$	

1.3. Librairies à utiliser

Les valeurs de X, Y, Z *et* ε ont une partie entière et une partie fractionnaire, donc au lieu d'utiliser la commande std_logic_vector, pour déclarer les variables, il faudra utiliser la commande sfixed (nombres signés). Cependant pour utiliser cette ligne de code il faudra utiliser le package « fixed_pkg », laquelle est inclut dans la norme VHDL 2008. Lors de la déclaration d'un objet de type sfixed il faut inclure le nombre de bits à utiliser pour les parties entières et fractionnaires. La partie fractionnaire sera située à droite de l'indice 0. Un avantage sur le type numeric_std c'est que sur fixed_pkg, le débordement n'est pas possible puisqu'un bit est explicitement ajouté (Langlois, 2013).

1.4. Cahier des charges

Les besoins exprimés sont :

- Développer un programme pouvant faire le calcul Cordic.
- Créer les fichiers sources des composants.
- Créer les fichiers sources des tests des composants.
- Écrire une documentation pour suivre le développement.

1.5. Exigences

D'après le cahier des charges, les exigences ont été produites et reportées dans le tableau 2

Tableau 2. Exigences du système Cordic

Exigences	Description
E1	Logiciel à utiliser : Model Sim
E2	Taille des vecteurs X *et* Y: 1 bit de signe, 1 bit partie entière, 14 bits partie fractionnaire
E3	Z *et* ε: 1 bit de signe, 2 bits partie entière, 13 bits partie fractionnaire
E4	14 valeurs de ε stockées dans une mémoire ROM.
E5	Création d'une machine avec les états : Repos, ChargeReg, Fin, iterations.
E6	Calculer les résultats en mode Rotation et vectorisation
E7	Documentation du système

2. Développement

2.1. Schéma général du système

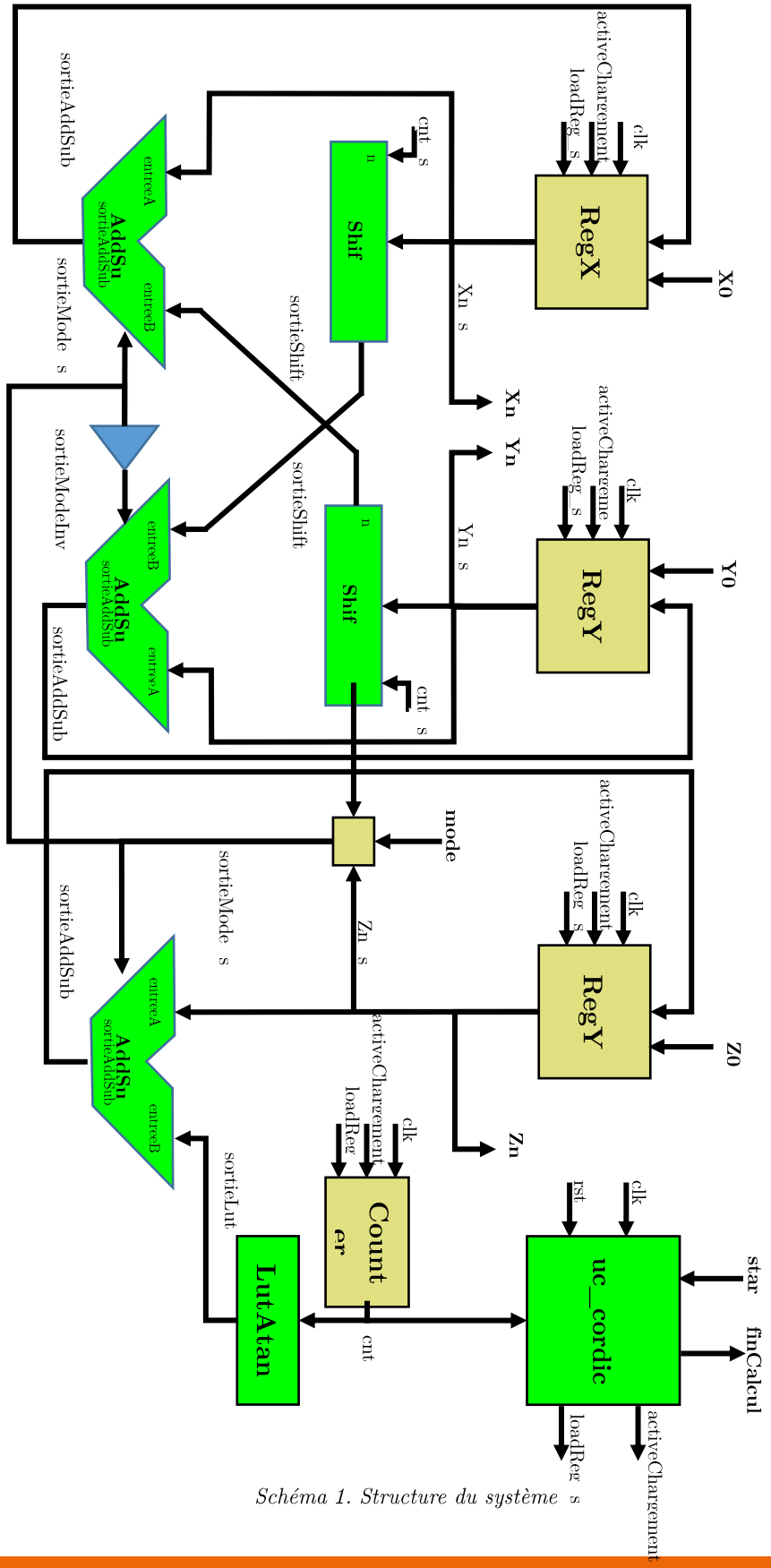


Schéma 1. Structure du système

2.2. Composants du système

Le système sera composé de sept éléments principaux :

a. Registre

Le but de cet élément est de choisir quelles données doivent être utilisées sur le système. Le système Cordic part d'une valeur initiale introduite par l'utilisateur « **E__Reg__0** », puis cette valeur change à chaque cycle d'itération « **E__Reg__I** ». Ce sera la valeur utilisée par le système.

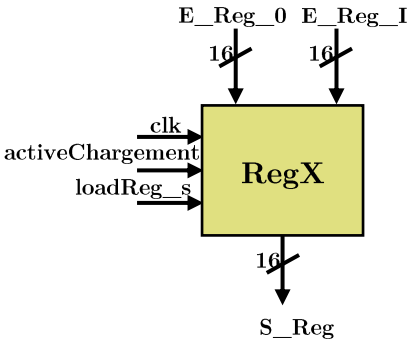


Schéma 2. Composant Registre

Cependant, le composant ne fonctionnera que si la valeur « **loadReg__S** » est égale à '1'. Egalement que l'entrée « **E__Reg__I** » soit prise en compte, la valeur de « **activeChargement** » devra être égale à '1'. Comme une valeur antérieure sera stockée, il est important que l'information soit synchronisée avec un système de référence de temps. Le système aura donc un signal « **clk** ». Finalement le composant aura une sortie « **S__Reg** » qui sera égale à la valeur initiale ou à la valeur du registre Le composant registre permettra d'utiliser les valeurs initiales (X_0, Y_0, Z_0) et de sauvegarder les valeurs X_i, Y_i, Z_i afin d'avoir les données suffisantes pour utiliser les formules suivantes : $X_{i+1} = X_i - Y_i d_i 2^{-i}$, $Y_{i+1} = Y_i + X_i d_i 2^{-i}$ et $Z_{i+1} = Z_i - d_i \varepsilon_i$

b. Shift

Pour calculer X_{i+1} et Y_{i+1} les expressions suivantes seront utilisées :

$$X_{i+1} = X_i - Y_i d_i 2^{-i} \text{ et } Y_{i+1} = Y_i + X_i d_i 2^{-i}$$

Où 2^{-i} peut être traduit comme un déplacement arithmétique vers la droite, qui changerai en fonction du nombre d'itérations du système. « **cnt__s** » sera donc le nombre de déplacements à faire et « **S__shiftn** » sera la sortie du système du signal déplacé.

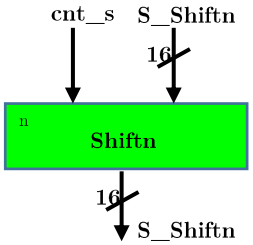


Schéma 3. Composant Shiftin

c. Additionneur/Soustracteur (AddSub)

Pour calculer X_{i+1}, Y_{i+1} , Z_{i+1}, deux opérations arithmétiques pourront se faire l'addition et la soustraction. La valeur « **add__sub** » définit l'opération qui sera effectuée (1 addition / 0 soustraction). La sortie du composant AddSub, « **S__addsub** » sera l'opération arithmétique entre les deux signaux « **E__addsub__1** » et « **E__addsub__2** ».

Il faut préciser que les valeurs de X_i, Y_i, Z_i seront mises dans l'entrée A et les valeurs $Y_i 2^{-i}$, $X_i 2^{-i}$, ε_i, dans l'entrée B.

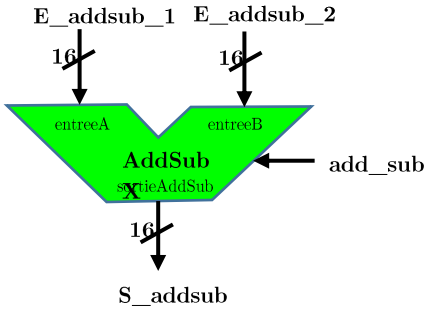


Schéma 4. Composant AddSub

Remarque : Lors de l'utilisation de la bibliothèque « fixed_pkg », lorsqu'une opération d'addition ou de soustraction est effectuée, un bit complémentaire est ajouté pour éviter le débordement, cependant pour maintenir le même dimensionnement vectoriel, la valeur de l'opération arithmétique est stockée dans une variable et ensuite sectionnée pour sauvegarder le signe numérique et le nombre avec le dimensionnement correct.

d. Sélection des signes

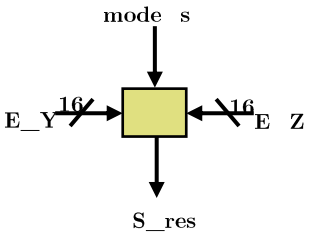


Schéma 5. Composant Signe

Pour déterminer quelle opération arithmétique sera effectuée sur la composante addition et soustraction. Il faudra considérer la valeur de d_i et le type d'opération arythmique de base. Si la valeur d'entrée introduite par l'utilisateur « **mode_S** » est égale à '0' le système travaillera en mode vectorisation et si c'est égale à '1' en mode rotation. Dans le cas de vectorisation, le bit le plus significatif de l'entrée « **E_Y** » est pris, puisqu'á cette position se trouve le signe ('0' positif et '1' négatif). Pour le cas de la rotation, il faudra prendre le bit plus significatif de l'entrée « **E_Z** ». Cependant si « **S_RES** » prend la valeur du bit plus significatif pour définir l'opération à faire, dans le cas rotation les expressions mathématiques seront incorrectes. Il suffira d'inverser la valeur du bit plus significative de « **E_Z** » pour corriger le problème.

Tableau 3. Remarques mathématiques

Rotation
$d_i \begin{cases} -1, & si\ Z_i < 0 \rightarrow MSB\ \ Z_i = 1 \rightarrow \begin{cases} X_{i+1} = X_i + Y_i 2^{-i} \\ Y_{i+1} = Y_i - X_i 2^{-i} \\ Z_{i+1} = Z_i + \varepsilon_i \end{cases} \rightarrow \ \overline{Z}_\iota = 0 \rightarrow \begin{cases} X_{i+1} = X_i - Y_i 2^{-i} \\ Y_{i+1} = Y_i + X_i 2^{-i} \\ Z_{i+1} = Z_i - \varepsilon_i \end{cases} \\ 1, & si\ Z_i \geq 0 \rightarrow MSB\ \ Z_i = 0 \rightarrow \begin{cases} X_{i+1} = X_i - Y_i 2^{-i} \\ Y_{i+1} = Y_i + X_i 2^{-i} \\ Z_{i+1} = Z_i - \varepsilon_i \end{cases} \rightarrow \ \overline{Z}_\iota = 1 \rightarrow \begin{cases} X_{i+1} = X_i - Y_i 2^{-i} \\ Y_{i+1} = Y_i + X_i 2^{-i} \\ Z_{i+1} = Z_i - \varepsilon_i \end{cases} \end{cases}$
Vectorisation
$d_i \begin{cases} -1, & si\ Y_i \geq 0 \ \rightarrow MSB\ \ Y_i = 0 \ \rightarrow \begin{cases} X_{i+1} = X_i - Y_i 2^{-i} \\ Y_{i+1} = Y_i + X_i 2^{-i} \\ Z_{i+1} = Z_i - \varepsilon_i \end{cases} \\ 1, & si\ Y_i < 0 \rightarrow MSB\ \ Y_i = 1 \rightarrow \begin{cases} X_{i+1} = X_i - Y_i 2^{-i} \\ Y_{i+1} = Y_i + X_i 2^{-i} \\ Z_{i+1} = Z_i - \varepsilon_i \end{cases} \end{cases}$
Attention : Pour calculer Y_{i+1} il faut inverser la valeur « **S_RES** », étant l'opération contraire de X_{i+1} et Z_{i+1},

e. Machine d'état Cordic (uc_cordic)

Ce composant contrôle tout le système avec quatre états : *'Repos'*, *'ChargeReg'*, *'Iterations'* et *'Fin'*. Pour effectuer de manière séquentielle chaque transition d'états, le signal d'entré « **clk** » sera utilisé. En cas d'arrêt et de redémarrage du système, il suffira de mettre à '1' la valeur d'entrée « **rst** ».

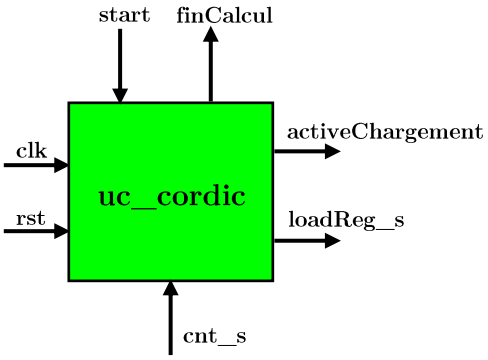


Schéma 6 uc_cordic

L'état de '*repos*' sera l'état initial du système. Si l'entrée « **Start** » est mise à '1' le système commencera à fonctionner sinon il restera sur le même état. Le deuxième état '*ChargeReg*' est un état de transition ou le composant Registre est activé lorsque la valeur « **loadReg_s** » est mise à '1'. Le troisième état '*Itérations*' active le composant Registre afin de prendre les données sauvegardées et le composant compteur pour effectuer des itérations lorsque la valeur « **activeChargement** » est mise à '1'. Le système restera en boucle à l'état '*Itérations*' pendant un certain nombre d'itérations. Le dernier état '*Fin*' mis à '1' la valeur de sortie « **finCalcul** » qui indique que le calcul est terminé. Pour effectuer un autre calcul et retourner à l'état de '*repos*', la valeur de « **Start** » sera remise à zéro.

f. Mémoire ROM (LutAtan)

Pour calculer Z_{i+1}, les valeurs de ε_i devront être connues. L'expression mathématique suivante donc est utilisée :

$$\varepsilon_i = arctg(2^{-i})$$

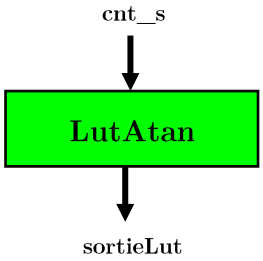


Schéma 7.Composant LutAtan

Comme ε_i aura i différentes valeurs, elles seront stockées dans une mémoire ROM. Les valeurs stockées seront en radians et en format bit. Si on accède à une adresse avec la valeur d'entrée « **cnts_s** » le nombre obtenu sera donc égale à la valeur de sortie « **sortieLut** » de la mémoire ROM. Les valeurs stockées dans la mémoire ont été calculées manuellement en format décimal et à l'aide de la commande "to_sfixed". Avec cette commande le nombre est transformé en bits (1 bit pour le signe, 2 bits pour la partie entière et 13 bits pour la partie fractionnaire).

g. Compteur

Le composant permettra d'effectuer un nombre défini d'itérations et il sera basé sur la valeur d'une horloge « **clk** ». Seulement si les entrées « **loadReg_s** » et « **activeChargement** » ont une valeur égale à '1', le compteur augmentera des itérations, sinon il restera sur le même nombre. Si « **loadReg_s** » et « **activeChargement** » ont une valeur égale à '0' le compteur sera remis à zéro.

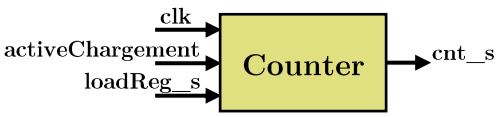


Schéma 8. Composant Counter

2.3. Tests des composants du système

Dans cette section, le test de fonctionnement de chaque composant constituant le système Cordic sera expliqué. Pour effectuer ces tests de fonctionnement, il faudra :

- Définir la taille du vecteur avec lequel on va travailler.
- Déclarer l'information du composant à tester.
- Déclarer les signaux à utiliser.
- Instancier les composants à tester.
- Développer les tests à réaliser.

a. Registre

Le test consistera à simuler trois cas : Le premier cas où l'élément Registre n'est pas activé, le deuxième cas ou la sortie du système sera égale à la valeur de « **REG_0** » et finalement le dernier cas ou la sortie du système sera égale à la valeur de « **REG_I** ».

```
        begin
                SLoadReg_s<='0';          --Cas où l'élément Registre n'est pas activé
                Sactivechargement<='0';
                SE_REG_0<="1100000000000000";
                SE_REG_I<="1111000000000000";
                wait for 20 ns;
                SLoadReg_s<='0';         --Cas dans lequel la sortie du système doit être le signal REG_0
                SE_REG_0<="0001000000000000";
                SE_REG_I<="1111000000000000";
                wait for 20 ns;
                SLoadReg_s<='1';         --Cas dans lequel la sortie du système doit être le signal REG_I
                SE_REG_0<="0001000000000000";
                SE_REG_I<="1111000000000000";
                wait for 20 ns;
        end process;
end architecture;
```

Code 1. Test composant Registre

- Cas 1 : « **SLoadReg_s** » est égal à '0', il n'y pas de valeur de sortie, puisque le composant Registre est désactivée.
- Cas 2 : « **SLoadReg_s** » est égal à '1' et « **ActiveChargement** » est égal à '0', la valeur de sortie est donc égale à « **SE_Reg_0** ».
- Cas 3. « **SLoadReg_s** » est égal à '1' et « **ActiveChargement** » est égal à '1', la valeur de sortie est donc égale à « **SE_Reg_I** ».

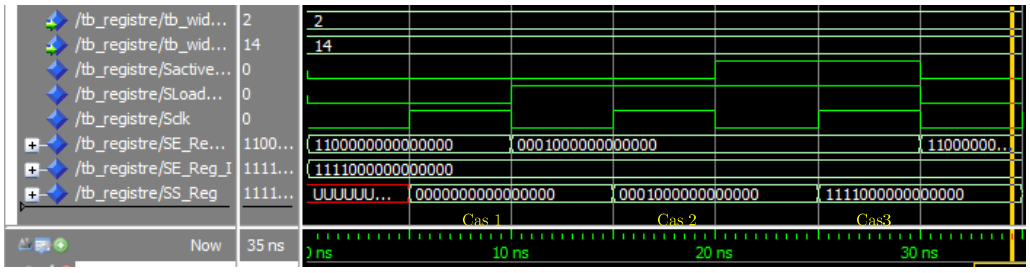


Figure 1. Comportement du composant Registre

b. Shift

Le test consistera à simuler deux cas : Le premier cas effectuera un décalage de quatre positions d'un nombre positif. Le second cas effectuera un décalage de deux positions mais d'un nombre négatif.

```
        process
        begin
                Scnt_s<= 4; --Decalage vers droit de quatre positions
                SE_Shiftn<="0100000000000000";
                wait for 20 ns;

                Scnt_s<= 2; --Decalage vers droit de deux positions
                SE_Shiftn<="1010000000000000";
                wait for 20 ns;
        end process;
end tb_test;
```

Code 2. Test du composant Shiftn

Pour le premier cas un décalage simple est fait, mais quand le nombre est négatif, le décalage conserve le signe puis il effectue le décalage arithmétique attendu.

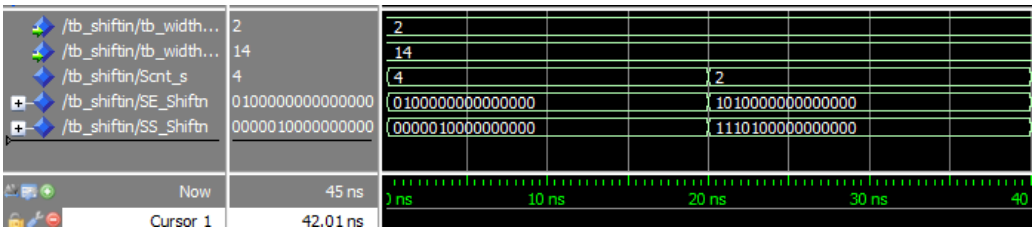


Figure 2. Comportement du composant Shiftn

c. Additionneur/Soustracteur (AddSub)

Afin de vérifier le fonctionnement du composant, quatre cas seront effectués. Les deux premiers seront des additions et les autres seront des soustractions.

```
        process
        begin
                Sadd_Sub<='0';
                SEntree_1<="0010000000000000"; -- Adittion 0.5+0.5=1.0
                SEntree_2<="0010000000000000"; -- C'est a dire "0100000000000000"
                wait for 10 ns;
                SEntree_1<="0100000000000000"; -- Adittion 1.0+0.75=1.75
                SEntree_2<="0011000000000000"; -- C'est a dire "0111000000000000"
                wait for 10 ns;
                Sadd_Sub<='1';
                SEntree_1<="0100000000000000"; -- Sustraction  1.0-0.125=0.875
                SEntree_2<="0000100000000000"; -- C'est a dire "0011100000000000"
                wait for 10 ns;
                SEntree_1<="0111000000000000"; -- Sustraction 1.75-0.25= 1.5
                SEntree_2<="0001000000000000"; -- C'est a dire "0110000000000000"
                wait for 10 ns;
        end process;
end architecture;
```

Code 3. Test du composant AddSub

Les résultats de la simulation correspondent aux résultats attendus :

- Cas 1 : L'entrée SEntree_1=0.5 et l'entrée SEntree_ 2=0.5, le résultat de l'addition est

SS_addsub =1.0 qui corresponde à la sortie du système.

- Cas 2 : L'entrée SEntree_1=1.0 et l'entrée Sentree_ 2=0.75, le résultat de l'addition est SS_addsub =1.70 qui correspond à la sortie du système.
- Cas 3 : L'entrée SEntree_1=1.0 et l'entrée Sentree_ 2=0.125, le résultat de la soustraction est SS_addsub =0.875 qui correspond à la sortie du système.
- Cas 4 : : L'entrée SEntree_1=1.75 et l'entrée Sentree_ 2=0.25, le résultat de la soustraction SS_addsub =1.75 qui correspond à la sortie du système.

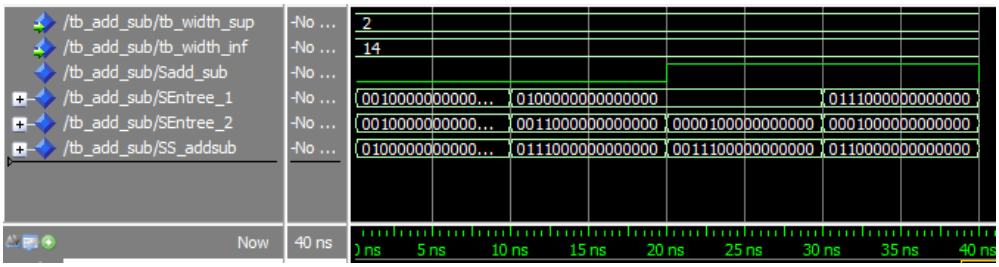


Figure 3. Comportement du composant AddSub

d. Sélection des signes

Pour vérifier le fonctionnement du composant, quatre cas seront effectués. Les deux premières représenteront le mode vectorisation et les autres le mode rotation.

```
process
        begin
                Smode_s<='0'; --vectorisation
                SE_Y<="0000000000000000"; --La sortie doit être le bit
                SE_Z<="1000000000000000"; -- le plus significatif de Y
                wait for 15 ns;
                SE_Y<="1000000000000000";
                SE_Z<="0000000000000000";
                wait for 15 ns;
                Smode_s<='1'; --Rotation
                wait for 15 ns;
                SE_Z<="1000000000000000"; --La sortie doit être l'inverse du bit
                SE_Y<="0000000000000000"; -- le plus significatif de Z
                wait for 15 ns;
        end process;
end tb_test;
```

Code 4. Test du composant Signe_Sigma

Les résultats de la simulation correspondent aux résultats attendus :

- Cas 1 : Le bit plus significatif de SE_Y= '0' et le bit plus significatif de SE_Z= '1', alors, en mode vectorisation, la sortie SS_res = SE_Y= '0'.
- Cas 2 : Le bit plus significatif de SE_Y= '1' et le bit plus significatif de SE_Z= '0', alors, en mode vectorisation, la sortie SS_res = SE_Y= '1'.
- Cas 3 : Le bit plus significatif de SE_Y= '1' et le bit plus significatif de SE_Z= '0', alors, en mode rotation, la sortie SS_res = $\overline{\text{SE\_Z}}$= '1'.
- Cas 5 : Le bit plus significatif de SE_Y= '0' et le bit plus significatif de SE_Z= '1', alors, en mode rotation, la sortie SS_res = $\overline{\text{SE\_Z}}$= '0'.

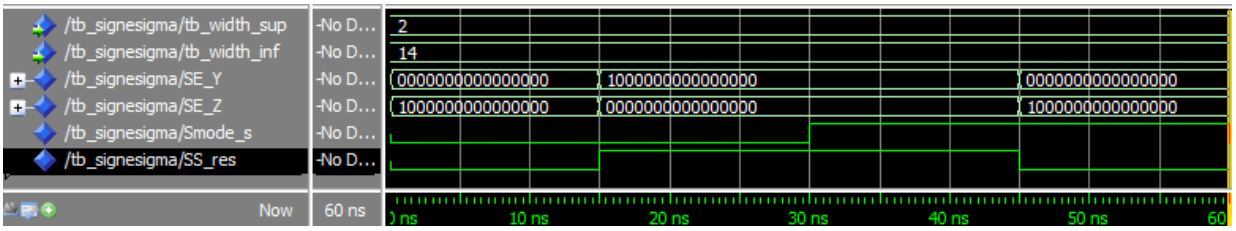


Figure 4. Comportement du composant Signe_Sigma

e. Machine d'état Cordic (uc_cordic)

Pour vérifier le fonctionnement du composant, le système devra passer par les quatre états de la machine d'états. Il devra aussi revenir à l'état de repos.

```
        process
        begin
                Srst<='0';
                Sstart<='1';  -- Début de la séquence
                wait for 200 ns; --Pendant ce temps, les quatre
                                 --états de la machine d'état seront exécutés.
                Sstart<='0'; --Retour à l'état de repos
                wait for 20 ns;
        end process;
end tb_test;
```

Code 5. Test du composant uc_cordic

La figure suivante montre le changement des états. L'état de basse du système est *'Repos', donc* pour commencer le calcul, **Sstar** se met à '1'. La machine d'état passe donc vers l'état *'ChargeReg'* où le composant Registre sera activé et immédiatement fera une transition vers l'état *'Iterations'*. Dans cet état la valeur **SactiveChargement** se met à '1'. Le composant compteur est activé et le composant Registre est habilité pour utiliser les valeurs précédemment enregistrées. Lorsque le compteur atteint la valeur limite, il passe à l'état '*Fin*' où tout le calcul s'arrête. **SfinCalcul** est mis à '1' afin d'avertir que le calcul est terminé. Le système restera à l'état '*Fin*" jusqu'à ce que la valeur **Start** soit remise à zéro pour retourner à l'état de '*Repos*'.

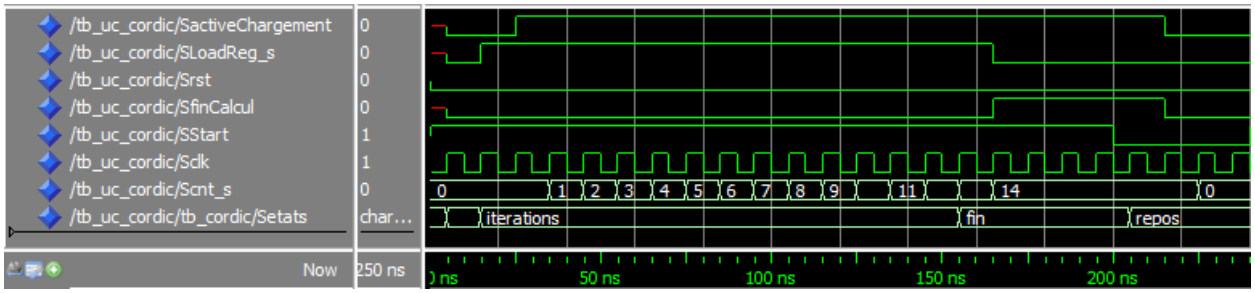


Figure 5. Comportement du composant uc_cordic

f. Mémoire ROM (LutAtan)

Pour vérifier le fonctionnement du composant, il faudra simuler différentes valeurs des adresses pour examiner que toutes les données de la ROM ont été correctement accédées.

```
        process
        begin
                --Affichage des valeurs dans la fenêtre de commande
                report real'image(to_real(SSortieLut));
                wait for 10 ns;
                --Compteur pour accéder aux valeurs de la mémoire rom
                Scnt_S<=Scnt_s+1;
        end process;
end tb_test;
```

Code 6. Test du composant LutAtan

L'accès à chaque adresse de la ROM est contrôlé par la valeur du compteur, ce qui implique que lorsque cette valeur change, l'adresse change également. Ce comportement est représenté par les figures suivantes.

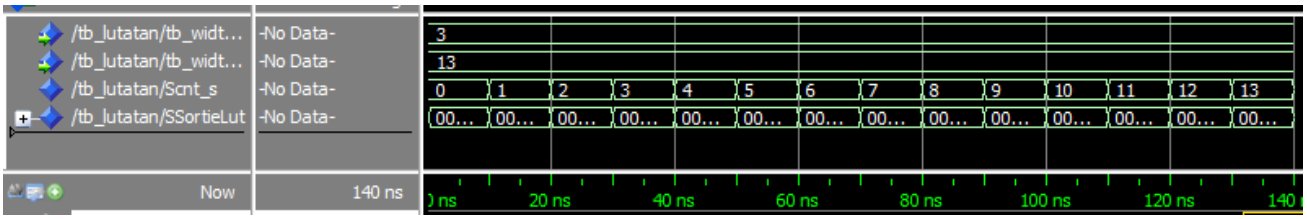


Figure 6. Comportement du composant LutAtan

```
# ** Note: 7.854004e-01
#    Time: 10 ns  Iteration: 0  Instance: /tb_lutatan
# ** Note: 4.636230e-01
#    Time: 20 ns  Iteration: 0  Instance: /tb_lutatan
# ** Note: 2.449951e-01
#    Time: 30 ns  Iteration: 0  Instance: /tb_lutatan
# ** Note: 1.243896e-01
#    Time: 40 ns  Iteration: 0  Instance: /tb_lutatan
# ** Note: 6.237793e-02
#    Time: 50 ns  Iteration: 0  Instance: /tb_lutatan
# ** Note: 3.125000e-02
#    Time: 60 ns  Iteration: 0  Instance: /tb_lutatan
# ** Note: 1.562500e-02
#    Time: 70 ns  Iteration: 0  Instance: /tb_lutatan
# ** Note: 7.812500e-03
#    Time: 80 ns  Iteration: 0  Instance: /tb_lutatan
# ** Note: 3.906250e-03
#    Time: 90 ns  Iteration: 0  Instance: /tb_lutatan
# ** Note: 1.953125e-03
#    Time: 100 ns  Iteration: 0  Instance: /tb_lutatan
# ** Note: 9.765625e-04
#    Time: 110 ns  Iteration: 0  Instance: /tb_lutatan
# ** Note: 4.882813e-04
#    Time: 120 ns  Iteration: 0  Instance: /tb_lutatan
# ** Note: 2.441406e-04
#    Time: 130 ns  Iteration: 0  Instance: /tb_lutatan
# ** Note: 1.220703e-04
#    Time: 140 ns  Iteration: 0  Instance: /tb_lutatan
```

Figure 7. Donnes de chaque adresse du composant LutAtan

g. Compteur

Pour vérifier le fonctionnement du composant, il faudra démarrer le compteur puis après un certain nombre d'itérations l'arrêter.

```
        process
        begin
                SactiveChargement<='1'; --Démarrage par compteur
                SLoadReg_s<='1';
                wait for 200 ns;
                SactiveChargement<='0'; --Arrêt du compteur
                SLoadReg_s<='1';
                wait for 10 ns;
        end process;
end tb_test;
```

Code 7. Test du composant Compteur

L'image suivante représente l'évolution du composant en fonction des interactions extérieures. Lorsque SLoadReg_**s** et **SactiveChargement** sont actifs, le composant commence à compter. Quand le compteur atteint le nombre 20 **SactiveChargement** change d'état et donc le compteur s'arrête. Dans le cas où le compte est mis à zéro, **SLoadReg_s** et **SactiveChargement** doivent tous les deux être dans l'état zéro.

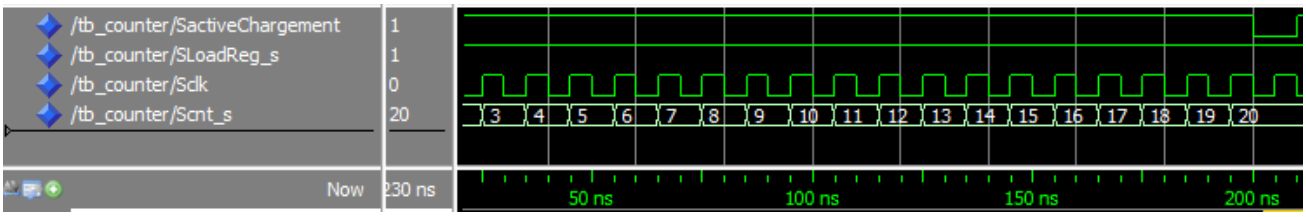


Figure 8. Comportement du composant Compteur

2.4. Intégration du système

a. Composant principal

Le système Cordic ressemble à un grand bloc qui a sept entrées et quatre sorties, comme le montre le schéma ci-dessous.

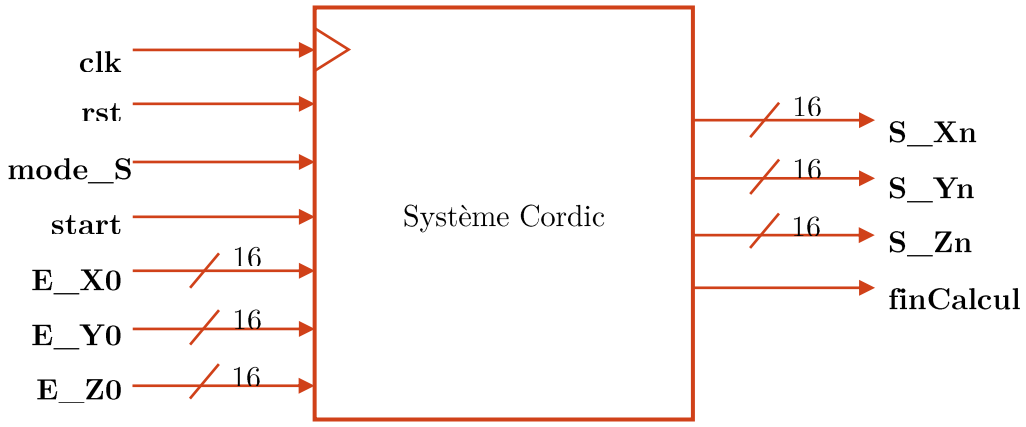


Schéma 9. Boite noire du composant Cordic

Entrées :

- clk : Signal d'horloge afin d'avoir un système synchrone.
- rst : Valeur d'entrée qui interrompt le système général et remet les variables à zéro.
- mode_S : Valeur qui détermine le mode d'opération du système Cordic, '0' pour vectorisation et '1' pour rotation.
- start : Valeur introduite par l'utilisateur afin de commencer le calcul et le réinitialiser.
- E_X0, E_Y0, E_Z0, : Valeurs initiales introduites par l'utilisateur.

Sorties :

- S_Xn, S_Yn, S_Zn, : Valeurs Cordic obtenues sur chaque cycle d'itération.
- finCalcul : Valeur de sortie qui indique quand le système Cordic est terminé.

A l'intérieur du système, 12 composants seront intégrés. Ils devront être instanciés afin de les inclure au sein du grand système. Les composants suivants seront occupés :

Tableau 4. Nombre total des composants à instancier

Composant	Quantité	Composant	Quantité
Registre	**3**	**Machine d'état Cordic**	**1**
Additionneur/Soustracteur	**3**	**Mémoire ROM**	**1**
Shift	**2**	**Compteur**	**1**
Sélection des signes	**1**		
		Total	**12**

Pour l'instanciation des composants, il faudra vérifier les liaisons des signaux d'entrée et de sortie du système. Il faudra déterminer quels signaux internes devront être créés afin de pouvoir partager des informations d'un composant à un autre. Certains points importants sont à souligner :

- **Registre :**
  - Puisque Z a un bit de signe, deux bits de partie entière et treize bits de partie fractionnaire, les limites supérieure et inférieure doivent être différentes des limites de X et Y.
- **Additionneur/Soustracteur (AddSub) :**
  - Puisque Z et **ε** ont un bit de signe, deux bits de partie entière et treize bits de partie fractionnaire, les limites supérieure et inférieure doivent être différentes des limites de X et Y.
  - Le type d'opération à effectuer sur Y sera toujours le contraire de X et Z, donc le signal doit être inversé. L'expression SS_resY<= not SS_res sera donc utilisée.
  - Puisque Z n'inclut pas de décalage dans la formule, il occupe la valeur de sortie du composant Mémoire ROM (LutAtan).
- **Mémoire ROM (LutAtan) :**
  - Puisque **ε** a un bit de signe, deux bits de partie entière et treize bits de partie fractionnaire, les limites supérieure et inférieure doivent être différentes des limites de X et Y.
- Les sorties (*S_Xn, S_Yn et S_Zn*) du système auront attribuées un signal interne afin d'utiliser ses valeurs entre les composants du système. Les relations suivantes : *S_Xn<=SS_Xn, S_Yn<=SS_Yn* et *S_Zn<=SS_Zn* seront donc utilisées.

b. Test de composant Cordic

Pour effectuer le test de fonctionnement du système, il faudra :

- Définir la taille de X_{i+1}, Y_{i+1} , Z_{i+1}et ε_i. C'est-à-dire, le bit de signe, la partie entière et la partie fractionnaire.
- Déclarer le composant Cordic.
- Déclarer les signaux à utiliser : Sclk, Srst, Sstart, SE_X0, SE_Y0, SE_Z0, SS_Xn, SS_Yn, SS_Zn et SfinCalcul.
- Réaliser l'instanciation des signaux.

Le système aura l'horloge en fonctionnement tout le temps. Les seules variables qui pourront être modifiées seront :

- Srst : Pour arrêter le programme et remettre tout à zéro.
- Sstart : Pour démarrer le calcul et aussi pour remettre le système dans l'état *Repos.*
- SE_X0, SE_Y0, SE_Z0 : Les valeurs introduites par l'utilisateur.

Les valeurs à monitorer seront SS_Xn, SS_Yn, SS_Zn et SfinCalcul. Avec la commande *report real'image(to_real(SS_Xn ou SS_Yn ou SS_Zn))* les valeurs seront en format décimal. La comparaison pourra donc être effectuée sur le site internet Online Cordic Calculator.

3. Résultats

3.1. Mode rotation

Les valeurs à tester en mode rotation sont :

- Xo=0.3125
- Yo=0.15625
- Zo=0.00097

Les résultats obtenus sur le système sont :

```
# ** Note: 5.139160e-02
#    Time: 210 ns  Iteration: 0  Instance: /tb_cordic
# ** Note: 2.575684e-02
#    Time: 210 ns  Iteration: 0  Instance: /tb_cordic
# ** Note: 0.000000e+00
#    Time: 210 ns  Iteration: 0  Instance: /tb_cordic
```

Figure 9. Résultats du Cordic en mode rotation

Les valeurs obtenues sur le site d'internet sont :

```
Decimal:
x[14] = 0.05143419899990595; y[14] = 0.025784675856044124; z[14] = -0.00007395777451643917
to Fixed Point Bin:
x[14] = 00.00001101001010; y[14] = 00.00000110100110; z[14] = 00.00000000000001
```

Figure 10. Résultat du Cordic sur le site d'internet Online Cordic Calculator.

3.2. Mode vectorisation

Les valeurs à tester en mode vectoristation sont :

- Xo=-0.3125
- Yo=0.15625
- Zo=0.00097

Les résultats obtenus sur le système sont :

```
# ** Note: 3.393555e-02
#    Time: 210 ns  Iteration: 0  Instance: /tb_cordic
# ** Note: 4.650879e-02
#    Time: 210 ns  Iteration: 0  Instance: /tb_cordic
# ** Note: 1.744141e+00
#    Time: 210 ns  Iteration: 0  Instance: /tb_cordic
```

Figure 11. Résultats du Cordic en mode vectorisation

```
Decimal:
x[14] = 0.03417576249598722; y[14] = 0.046285457690089495; z[14] = 1.7441411126599267
to Fixed Point Bin:
x[14] = 00.00001000101111; y[14] = 00.00001011110110; z[14] = 01.10111110100000
----------------------
```

Figure 12. Résultat du Cordic sur le site d'internet Online Cordic Calculator.

4. Conclusion

La valeur obtenue sur le système du Cordic reste une approximation puisque le système, au cours de ses itérations, ne prend pas les valeurs exactes mais des valeurs arrondies. De plus, lors de l'exécution d'un Shift arithmétique, on perd des valeurs importantes puisque les bits sont déplacés vers la droite. Comme le vecteur a une taille définie, il ne peut pas garder les bits en dehors de sa taille. A cause de ça, la précision du système est plus précise pour les cycles de 1 à 9, mais une légère marge d'erreurs apparaît lors des cycles suivants. Des solutions possibles sont d'augmenter la taille du vecteur en fonction du nombre d'itérations à faire ou travailler avec des nombres flottants et transformer, uniquement à la fin, le nombre flottant en bits. Toutefois, il y a un risque de faire un programme non synthétisable.

La réutilisation des composants est une façon d'améliorer le temps de développement d'une application puisque l'on ne repart pas de zéro. Cependant il faut prêter une attention particulière aux instanciations. En effet, plus notre système est grand, plus la quantité d'instanciations à faire sera grande. Cette augmentation se verra aussi dans la quantité de variables partagées entre les composants. La liste de sensibilité est très importante puisque l'application, même bien codée, peut donner des résultats erronés si un processus ne possède pas les entrées nécessaires à son bon fonctionnement. Analyser et déterminer les entrées à utiliser est donc nécessaire.

5. Références

Brunelle, É. (2015, 12 04). *Association mathématique du Québec*. Récupéré sur L'algorithme CORDIC: https://www.amq.math.ca/wp-content/uploads/bulletin/vol55/no4/09-maitre-CORDIC.pdf

Langlois, P. (2013). *Nombres binaires fractionnaires à virgule fixe: représentation et opérations*. Récupéré sur Polytechnique Montréal: https://moodle.polymtl.ca/pluginfile.php/89255/mod_folder/content/0/pdf/0602VirguleFixe.pdf?forcedownload=1

Wikipedia. (2018, 09 25). *CORDIC*. Récupéré sur Wikipedia: https://fr.wikipedia.org/wiki/CORDIC