KURKLU AHMET **GE4A-GROUPE 1**

TP ESN9

Conception de Systèmes Numériques

Implantation matérielle d'algorithme CORDIC

I-Introduction :

L'algorithme de CORDIC (COordinate Rotation Digital Computing) est une méthode simple et efficace pour le calcul d'une gamme de fonctions complexes. S'appuyant sur une technique de décalage addition et de rotation vectorielle, l'algorithme calcule notamment par approximation la plupart des fonctions de la trigonométrie. Beaucoup de calculatrices commerciales utilisent cet algorithme pour résoudre des fonctions telles que sinus, cosinus, arc tangente et racine carrée. Les systèmes de communication numériques l'utilisent également afin de produire rapidement des conversions polaires-cartésiennes sur des signaux´échantillonnées. Comme la plupart des algorithmes performants, l'efficacité de CORDIC repose sur sa simplicité et sa flexibilité. L'algorithme se compose en effet d'opérations mathématiques élémentaires ce qui, en plus d'offrir une grande efficacité de calcul, simplifie´énormément l'implantation sur des plateformes logicielles ou matérielles.
Le projet qui suit propose une brève expérience sur le design-flow complet de l'implantation matérielle d'un calculateur CORDIC. Partant d'un modèle en C de l'algorithme, il s'agit d'obtenir une description en VHDL synthétisable sur une cible de type FPGA.
Les objectifs de ce TP sont les suivants :

- illustrer la complémentarité entre les modèles comportementaux haut niveau (C) et les modèles utilisés pour la synthèse matérielle (VHDL),
- se familiariser avec les techniques d'implantation de fonctions numériques complexes en matériel (arithmétique en virgule fixe notamment),
- apprendre utiliser une chaine complète de développement pour FPGA.

II-Travail à effectuer

Le but du TP est la réalisation d'un circuit calculant le sinus et le cosinus d'un angle en utilisant l'algorithme présenté à la section précédente. Le circuit est délimité sur la Fig. 3. Son comportement temporel est explicité sur la Fig. 4.

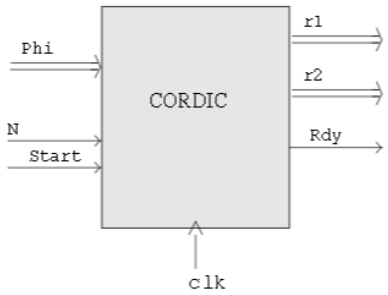


FIGURE 3 – Délimitation du circuit

La valeur de l'angle et le nombre d'itérations sont respectivement passés sur les entrées phi et n . La disponibilité de ces opérandes est signalée en passant à 1 le signal start . Le circuit accuse réception en passant rdy à 0. Les opérandes sont alors mémorisés en interne et le client peut repasser start à 0. La fin du calcul est signalée par la remontée de rdy à 1. Les résultats, cos(phi) et sin(phi), sont alors disponibles sur les sorties r1 et r2 respectivement. Le comportement est indéfini si start passe à 1 alors que rdy est à 0. Le signal rst permet de réinitialiser le circuit.

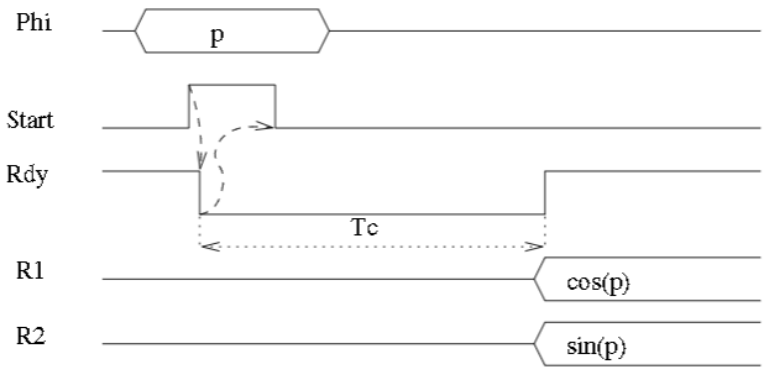


FIGURE 4 – Chronogrammes pour la spécification

II.1-Démarche à suivre

La démarche à suivre est d`es lors la suivante :

1. écriture et validation de l'algorithme en C avec des nombres en virgule flottante,
2. réécriture du modèle C afin qu'il opère sur des nombres codés en virgule fixe et validation,
3. traduction en VHDL du modèle C en virgule fixe, en utilisant un modèle comportemental et validation
4. transcription en un modèle VHDL architectural et simulation,
5. synthèse du code VHDL et estimation des performances

II.2- Modèle C en virgule flottante

En nous appuyant sur les principes donnés à la section 2 du sujet, nous avons écrit en C une fonction cordic(), qui calcule le sinus et le cosinus d'un angle. La fonction est la suivant vous pouvez trouver le code complet dans le répertoire V1-VirguleFloat_ErrCosSIn/main.c :

```c
void cordic(double phi, int n,double *x,double*y){
    int i;
    double alpha,teta,n_x,n_y,tmp_x,tmp_y,d;

    n_x= tabA[MAX-1];
    n_y = 0;
    d=1;
    alpha=0;
    tmp_x=0;
    tmp_y=0;
    teta=tabTeta[0];

    for(i=0;i<n;i++){
        if(alpha < phi){
            tmp_x = n_x - n_y * d;
            tmp_y = n_y + n_x * d;

            alpha = alpha + teta;
        }
        else{
            tmp_x = n_x +n_y * d;
            tmp_y = n_y -n_x * d;
            alpha = alpha - teta;
        }
    n_x =tmp_x;
    n_y = tmp_y;
    d=d/2;
    teta = tabTeta[i+1];
    }

    *x = n_x;
    *y = n_y;
}
```

La fonction prend en argument :
- phi qui est l'angle exprimé en degrés et codé en double.
- n qui est le nombre d'itérations.
- x et y en entrée/sortie afin de récupérer les résultats.

Nous avons aussi avant de réaliser la fonction cordic() crée deux tableaux de double afin de stocker les constantes θi et Ai. Les tableaux sont déclarés en variable globale puis rempli lors de l'exécution de la fonction initialisationTab() au début du main.

```
double tabTeta[MAX];
double tabA[MAX];

double RtoD(double radian){
    return radian*(180/PI);
}
double DtoR(double degre){
    return degre*(PI/180);
}

void initTabTeta(int n){
    for (int i = 0; i < n; i++)
    {
        tabTeta[i]=RtoD(atan(pow(2,-i)));
    }
}

void initTabA(int n){
    tabA[0] = cos(DtoR(tabTeta[0]));
    for(int i =1;i<n;i++){
        tabA[i]=tabA[i-1]* cos(DtoR(tabTeta[i]));
    }
}

void initialisationTab(){
    initTabTeta(MAX);
    initTabA(MAX);
}
```

Les fonctions RtoD() et DotR() permettent respectivement la conversion de radian vers des degrés et de degrés vers des radians.

II.2.1- Tests élémentaires

Afin de tester le fonctionnement de notre programme nous avons réaliser une fonction test() qui est la suivante :

```
void test(){
    printf("----------------------------------------\n");
    printf("------------PROGRAM DE TEST-------------\n");
    printf("----------------------------------------\n");


    double angle;
    double n,x,y;
    printf("Entree l'angle :");
    scanf("%lf",&angle);

    printf("Entree le nombre d'iteration:");
    scanf("%lf",&n);

    cordic(angle,n,&x,&y);

    printf("x=%f,y=%f \n\n",x,y);

    printf("----------------------------------------\n");
    printf("------------FIN DU TEST-----------------\n");
    printf("----------------------------------------\n");

}
```

La fonction demande l'angle et le nombre d'itération et affiche x et y, qui sont le cos(phi) et le sin(phi). Nous obtenons le résultat suivant pour 30 itérations sur l'angle 60 :

```
----------------------------------------
------------PROGRAM DE TEST--------------
----------------------------------------
Entree l'angle :60
Entree le nombre d'iteration:30
x=0.500000,y=0.866025

----------------------------------------
------------FIN DU TEST------------------
----------------------------------------
```

i	a (°)	θ (°)	x	y
0	0.00	+45.00	1.0000	0.0000
1	45.00	+26.56	0.7071	0.7071
2	71.56	-14.03	0.3162	0.9487
3	57.53	+7.12	0.5369	0.8437
4	64.65	-3.58	0.4281	0.9037
5	61.08	-1.79	0.4836	0.8753
...				
13	60.001	0.007	0.5000	0.8660

On compare les valeurs obtenues avec la figure 2 du sujet. On remarque que les valeurs obtenues pour x et y sont similaire aux valeurs indiquées par le tableau donc notre fonction cordic() fonctionne correctement.

II.2.2- Qualification

Afin de réaliser la première étape de la qualification pour tracer les courbe Esin(phi) et Ecos(phi), nous avons écrit la fonction suivante :

```c
void quali(int n){
    printf("----------------------------------------\n");
    printf("------PROGRAM DE TEST QUALI-------------\n");
    printf("----------------------------------------\n");


    double x;
    double y;
    double angle =0;
    double erreurCos=0;
    double erreurSin=0;

    FILE*f1;
    FILE*f2;
    f1 = fopen("cosinus.txt","w+");
    f2 = fopen("sinus.txt","w+");

    for(angle=0;angle<91;angle++){
        cordic(angle,n,&x,&y);

        //printf("X = %f Y = %f\n",x,y);
        //printf("cosX = %f cosY = %f\n",cos(DtoR(angle)),sin(DtoR(angle)));
        erreurCos = fabs(cos(DtoR(angle))-x);
        erreurSin = fabs(sin(DtoR(angle))-y);

        //printf("errCos = %f errSin = %f\n",erreurCos,erreurSin);

        fprintf(f1,"%.40f\n",erreurCos);
        fprintf(f2,"%.40f\n",erreurSin);

    }
    fclose(f1);
    fclose(f2);


    printf("----------------------------------------\n");
    printf("------------FIN DU TEST-----------------\n");
    printf("----------------------------------------\n");

}
```

Cette fonction nous génère deux fichiers de sortie cosinus et sinus et nous utilisons la scripte octave qui se trouve dans le répertoire V1-VirguleFloat_ErrCosSIn/script1.m afin de tracer les courbes. Nous traçons les courbes pour un nombre d'itérations égal à 4, 8,19,32 puis enfin 64. Nous obtenons les courbes suivantes, on observe à gauche l'erreur sur le cosinus et à gauche l'erreur sur le sinus.

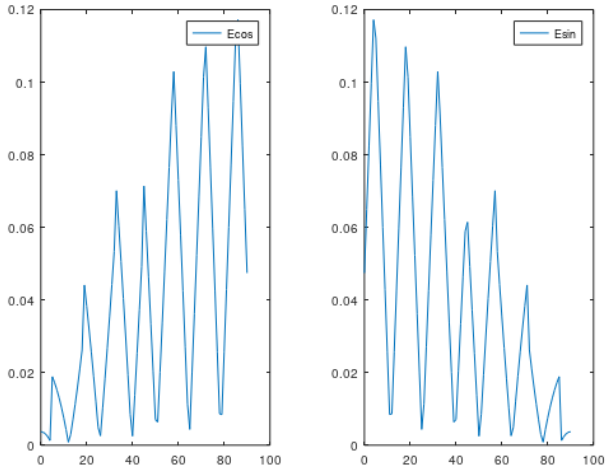


Figure 1 Erreur cosinus et sinus pour n = 4

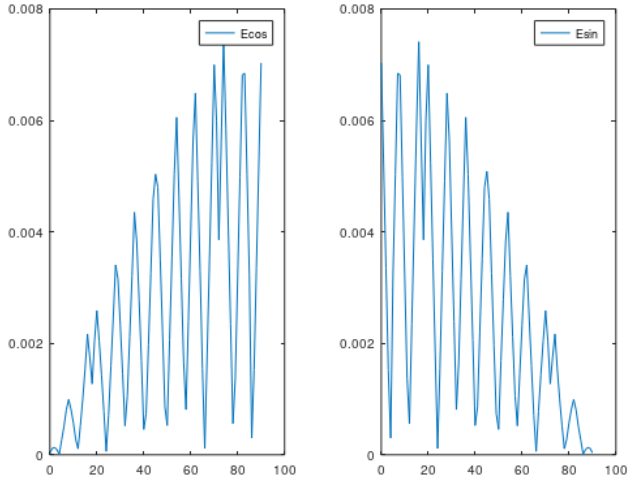


Figure 2 Erreur cosinus et sinus pour n = 8

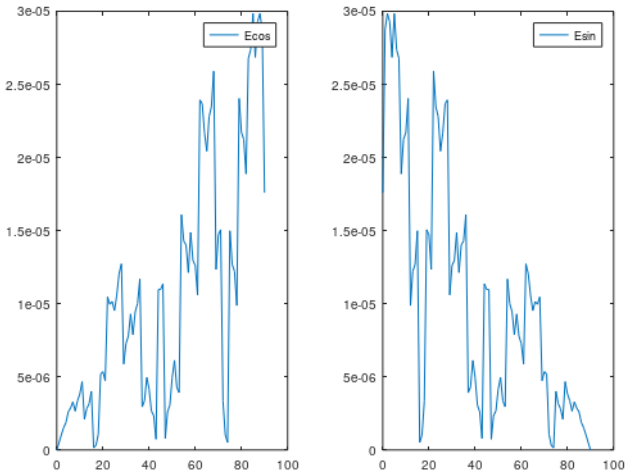


Figure 3 Erreur cosinus et sinus pour n = 16

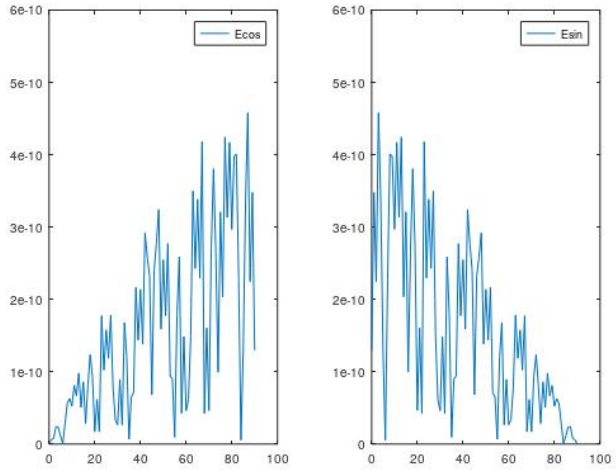


Figure 4 Erreur cosinus et sinus pour n = 32

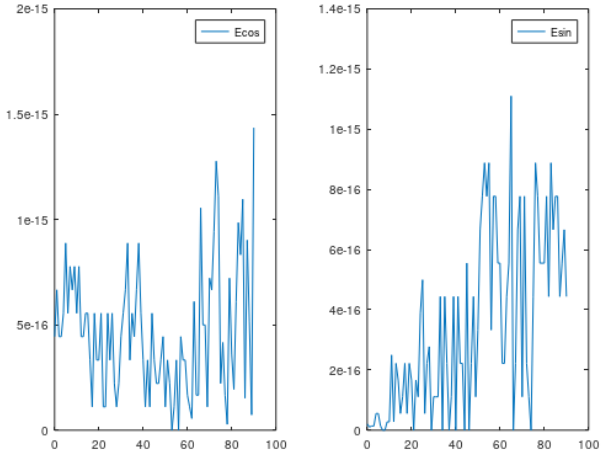


Figure 5 Erreur cosinus et sinus pour n = 64

On observe pour n = 4,8,16 et 32, que les courbes entre les cosinus et le sinus sont symétriques. Cela est dû à l'algorithme, car lorsque alpha > phi, on additionne le sinus et on soustrait le cosinus et lorsque alpha < phi, on soustrait le sinus et on additionne le cosinus. On obtient donc les mêmes courbes, mais inversés. Pour n = 64, on remarque que l'on perd la symétrie qui peut être expliquée par le bruit lié à la quantification.

On remarque de plus que l'erreur sur le cosinus et le sinus diminue avec l'augmentation du nombre d'itérations. Pour n=4 l'erreur maximale et d'environ 0.12, pour n = 8 l'erreur est de 0.008, pour n=32 l'erreur est d'environ 10^{-10} et enfin pour n = 64, on observe une erreur maximale d'environ 10^{-15}.

On détermine la précision en fonction du nombre d'itération à l'aide des courbes :

- Pour n = 4, on observe une précision de 10^{-3}.
- Pour n = 8, on observe une précision de 10^{-4}.
- Pour n = 16, on observe une précision de 10^{-6}.
- Pour n = 32, on observe une précision de 10^{-11}.
- Pour n = 64, on observe une précision de 10^{-32}.

Nous avons maintenant modifié notre programme afin de pouvoir tracer l'erreur maximale sur le sinus et le cosinus pour plusieurs valeurs de n. On obtient la fonction suivante :

```
void quali(int n,double *MaxC,double *MaxS){
    printf("----------------------------------------\n");
    printf("------PROGRAM DE TEST QUALI-------------\n");
    printf("----------------------------------------\n");


    double x;
    double y;
    double angle =0;
    double erreurCos=0;
    double erreurSin=0;

    double MaxErreurCos=0;
    double MaxErreurSin=0;


    for(angle=0;angle<91;angle++){
        cordic(angle,n,&x,&y);

        erreurCos = fabs(cos(DtoR(angle))-x);
        erreurSin = fabs(sin(DtoR(angle))-y);

        if(erreurCos>MaxErreurCos){
            MaxErreurCos = erreurCos;
        }
        if(erreurSin>MaxErreurSin){
            MaxErreurSin = erreurSin;
        }
    }

    *MaxC=MaxErreurCos;
    *MaxS=MaxErreurSin;


    printf("----------------------------------------\n");
    printf("------------FIN DU TEST-----------------\n");
    printf("----------------------------------------\n");

}
```

En utilisant les fichiers générer par la fonction quali() et le script octave dans le répertoire V2-VirguleFloat_ErrMax/script2.m on trace l'erreur maximale linéaire et l'erreur maximal logarithmique .

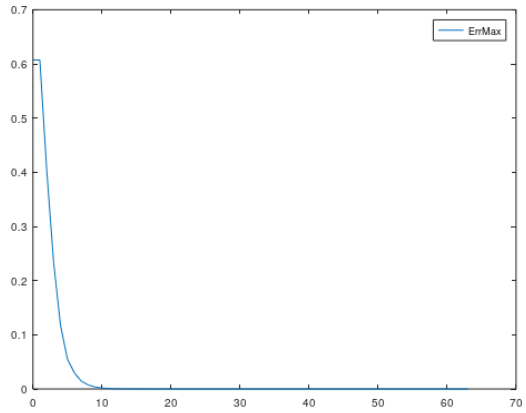


Figure 6 Erreur maximale cos/sin linéaire

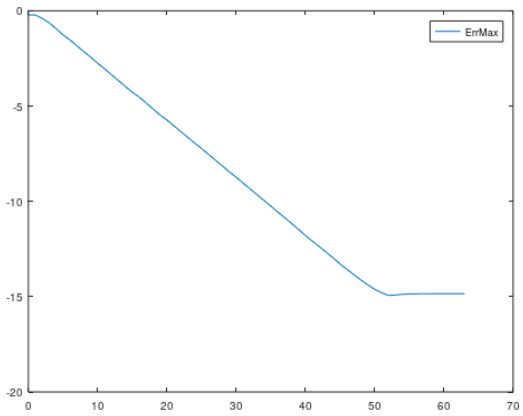


Figure 7 Erreur maximale cos/sin logarithmique

La courbe linéaire sur l'erreur maximale confirme que plus le nombre d'itérations augmente, plus l'erreur diminue. À l'aide de la courbe logarithmique, on remarque qu'après un certain point l'erreur est fixe cela signifie qu'il est inutile de faire plus d'itération après avoir dépassé ce point qui est n = 52.

II.3- Modèle C en virgule fixe

Le principe du modèle en virgule fixe est de remplacer les flottants par des entiers est d'effectuer uniquement des opérations d'addition, de soustraction et des décalages. Afin de réaliser cela, on doit d'abord définir des coefficients qui vont nous servir a :

- Obtenir des entiers en multipliant nos flottants par le coefficient.
- Obtenir des flottants en divisant nos entiers pas le coefficient.

En supposant que le codage se fasse sur des entier 16bits :
Les valeurs du cosinus et du sinus sont contenues entre -1 et 1, on a donc un pas de 2. On sait que 16 bits nous permettent de stocker 65 536 nombres. Si on divise 16 bits par 65536 on obtient 65536/2 = 32 768 , ont arrondie cette valeur à 32 000 afin d'être sûr de ne pas sortir de l'intervalle. Le coefficient pour le cosinus et le sinus est donc 32000.

La valeur des angles sont contenues a priori entre -90 et 90 degrés, mais lors des tests avec le modèle a virgules flottantes, on a remarqué des valeurs intermédiaires supérieure aux angles donnés dans le sujet. Nous allons donc supposer que les angles varient entre -120 et 120 degrés. Si on applique le même raisonnement, on obtient 65536/240 = 273, ont arrondi cette valeur à 270. Le coefficient pour nos angles est donc 270.

Avec les coefficients déterminer, on obtient pour le cosinus et le sinus un pas de 1/32000 = $3.125 * 10^{-5}$, et un pas de 1/270 = $3.703 * 10^{-3}$ pour les angles. On peut estimer une précision de 10^{-5} ce qui nous rapproche du cas avec n = 16 pour le modèle en virgule flottante. Nous allons donc supposer, que l'on a besoin de 16 itérations.

Afin de faciliter le codage et le décodage nous écrivons deux fonctions fp2fix() et fix2fp() le code complet du modèle en virgule fixe se trouve dans le répertoire V3-VirguleFIxe_ErrCosSin/main.c.

```
short int fp2fix(double M,double x){
    return floor(M*x);
}

double fix2fp(double M, short int x){
    return x/M;
}
```

La nouvelle fonction cordic() obtenue en appliquant les modifications est la suivante :

```c
void cordic2(short int phi, int n,short int *x,short int*y){
    int i;
    short int alpha,teta,n_x,n_y,tmp_x,tmp_y;

    n_x= tabA_fixe[MAX-1];
    n_y = 0;
    alpha=0;
    tmp_x=0;
    tmp_y=0;
    teta=tabTeta_fixe[0];

    for(i=0;i<n;i++){
        if(alpha < phi){
            tmp_x = n_x - (n_y>>i);
            tmp_y = n_y + (n_x>>i);

            alpha = alpha + teta;
        }
        else{
            tmp_x = n_x + (n_y>>i);
            tmp_y = n_y -(n_x>>i);
            alpha = alpha - teta;
        }
        n_x =tmp_x;
        n_y = tmp_y;
        teta = tabTeta_fixe[i+1];

        printf("Fixe Pour n =%d | x = %hu y = %hu\n",i,n_x,n_y);
        printf("NONF Pour n =%d | x = %f y = %f\n",i,fix2fp(M_CS,n_x),fix2fp(M_CS,n_y));

    }

    *x = n_x;
    *y = n_y;
}
```

Nous testons cette nouvelle fonction pour un angle de 60 et 16 itérations :

```
Fixe Pour n =15 | x = 15998 y = 27713
NONF Pour n =15 | x = 0.499937 y = 0.866031
```

On remarque encore une fois que les valeurs sont proches des valeurs données dans le sujet donc la fonction fonctionne correctement.

II.3.1- Qualification

En procédant de la même façon que pour la méthode à virgule flottante, on obtient les courbes suivantes :

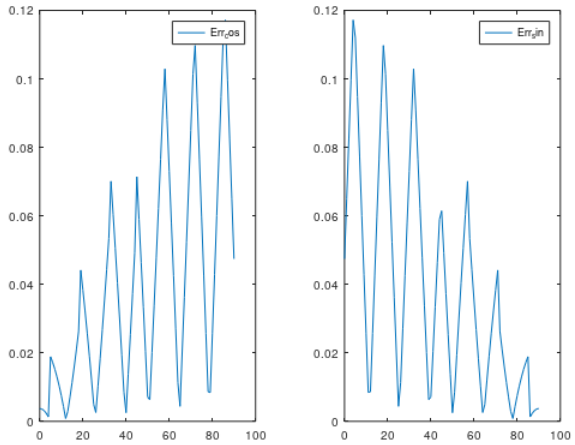


Figure 8 Erreur cosinus et sinus pour n = 4

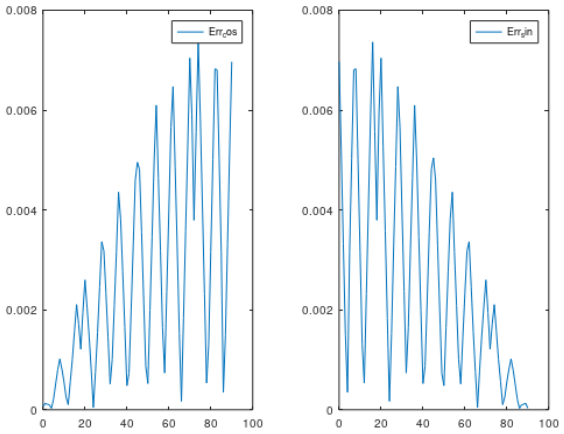


Figure 9 Erreur cosinus et sinus pour n = 8

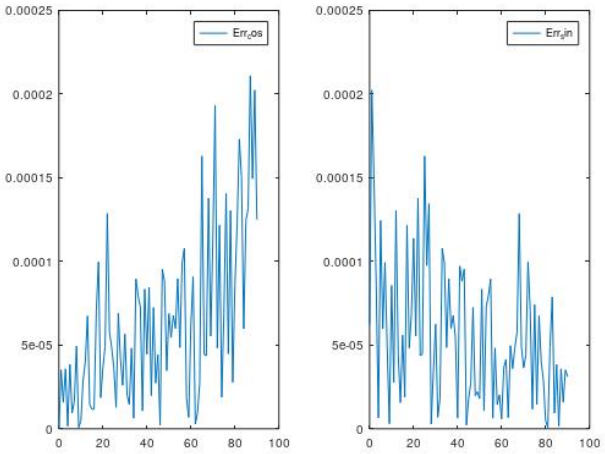


Figure 10 Erreur cosinus et sinus pour n = 16

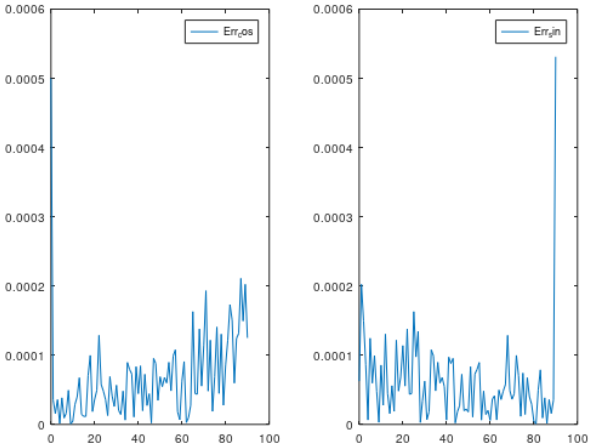


Figure 11 Erreur cosinus et sinus pour n = 32

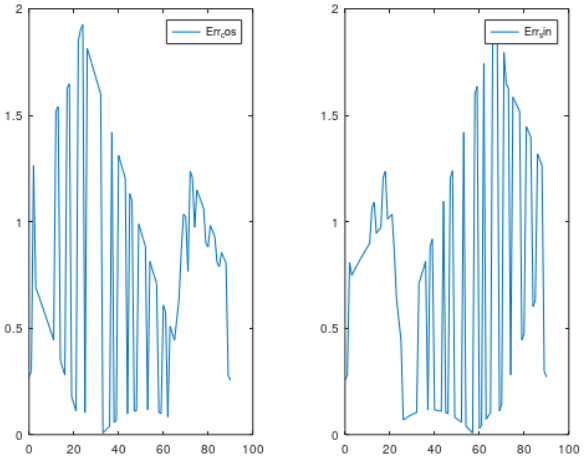


Figure 12 Erreur cosinus et sinus pour n = 64

On observe comme pour la méthode à virgule flottante que les courbes sont symétriques, que certaines des courbes sont symétrique. Cette symétrie s'interrompt cette fois pour n = 32. On remarque aussi que l'erreur diminue entre n=4 et n=16 puis augmente à nouveau entre n=16 et n=64. L'erreur qui augmente après un certain nombre d'itération peut s'expliquer à cause du type que l'on utilise. Nous utilisons des shorts, lorsque la précision augmente après n=16 nous avons besoin de plus de chiffres après la virgule, mais avec les coefficients que nous avons choisis, nous ne pouvons pas stocker ces nombres. Cela fait donc augmenter l'erreur. Au niveau de la précision, on remarque que pour n =4,8 et 16 la précision est identique au modèle à virgule flottante, mais arriver a 10^{-5} la précision ne diminue plus. On peut en conclure que le modèle à virgule fixe est moins précis.

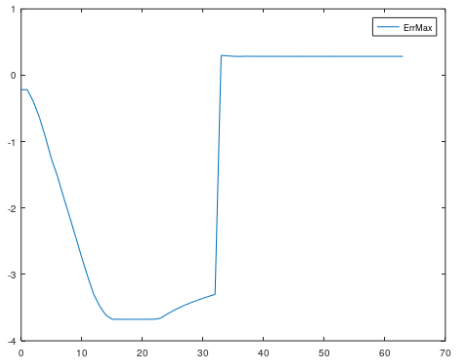


Figure 13 Erreur maximale cos/sin linéaire

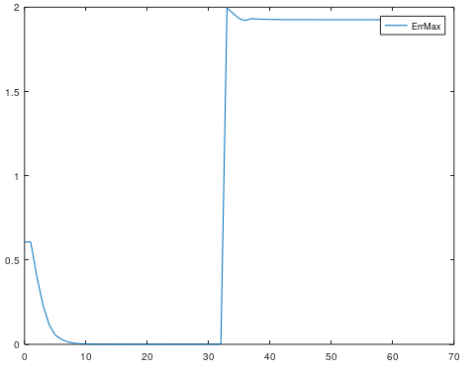
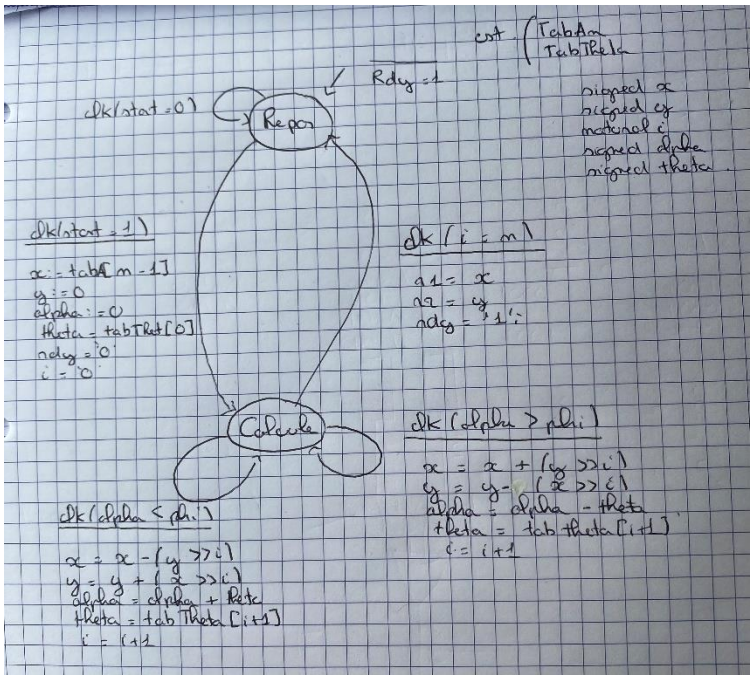


Figure 14 Erreur maximale cos/sin logarithmique

On remarque à l'aide de la courbe de l'erreur maximale linéaire et logarithmique qu'après environ 15 itérations l'erreur se stabilise, mais lorsque l'on dépasse 30 itérations l'erreur devient très grand. On en conclut donc qu'il est inutile de dépasser 16 itérations.

II.4- Modèle VHDL comportemental

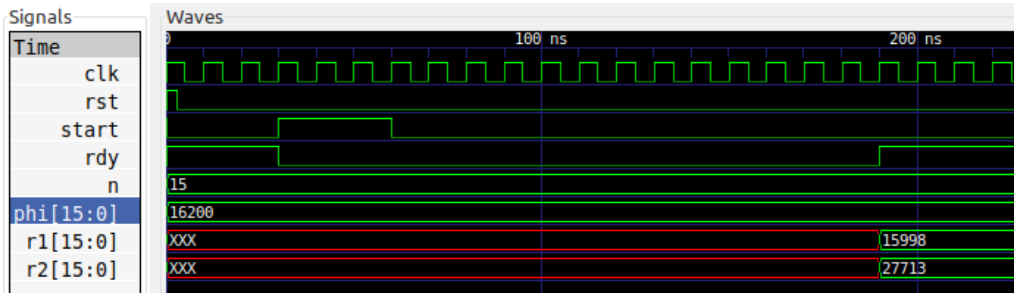
L'automate RTL que l'on obtient en s'inspirant des travaux faits en TD et de l'algorithme développé à la section précédente est le suivant :



La traduction de cet automate RTL en VHDL ainsi que le testbench pour effectuer les tests suivants sont disponible dans le répertoire CORDIC/.

Nous allons maintenant tester notre code VHDL avec 2 angles et comparé les valeurs obtenues avec notre modèle a virgule fixe.

Pour un angle de 60°, on obtient à l'aide de gtkwave les chronogrammes suivants :

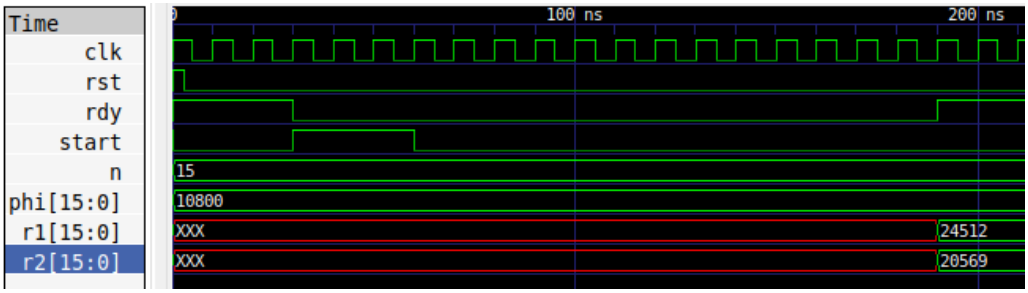


Si on converti les valeurs de r1 et r2 en flottant avec notre coefficient, on obtient r1 =15998/3200 = 0.4999375 et r2 =27713/3200 = 0.86603125

```
Fixe Pour n =15 | x = 15998 y = 27713
NONF Pour n =15 | x = 0.499937 y = 0.866031
```

On remarque que les valeurs obtenues par le code VHDL et le modèle a virgule fixe sont identique. Nous allons maintenant tester pour un angle de 40°.

On obtient les chronogrammes suivants :



Si on converti les valeurs de r1 et r2 en flottant avec notre coefficient, on obtient r1 =24512/3200 = 0.766 et

r2 =20569/3200 = 0.64278125

```
Fixe Pour n =15 | x = 24512 y = 20569
NONF Pour n =15 | x = 0.766000 y = 0.642781
```

On obtient encore une fois les mêmes valeurs avec le code VHDL et le modèle a virgule fixe. Le code VHDL fonctionne correctement.

Nous allons maintenant réaliser la synthèse du code VHDL sur Quartus :

Flow Status	Successful - Wed Mar 30 12:26:03 2022
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	CORDIC
Top-level Entity Name	CORDIC
Family	MAX 10
Device	10M50DAF484C6GES
Timing Models	Preliminary
Total logic elements	311 / 49,760 (< 1 %)
Total registers	100
Total pins	57 / 360 (16 %)
Total virtual pins	0
Total memory bits	0 / 1,677,312 (0 %)
Embedded Multiplier 9-bit elements	0 / 288 (0 %)
Total PLLs	0 / 4 (0 %)
UFM blocks	0 / 1 (0 %)
ADC blocks	0 / 2 (0 %)

À l'aide du rapport de synthèse généré par Quartus, on remarque que l'on utilise 57 pins, 100 registres pour un total de 311 blocs logiques. Pure les fréquences de fonctionnement max, on obtient :

Slow 1200mV 0C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	176.96 MHz	176.96 MHz	clk	

Slow 1200mV 85C Model Fmax Summary

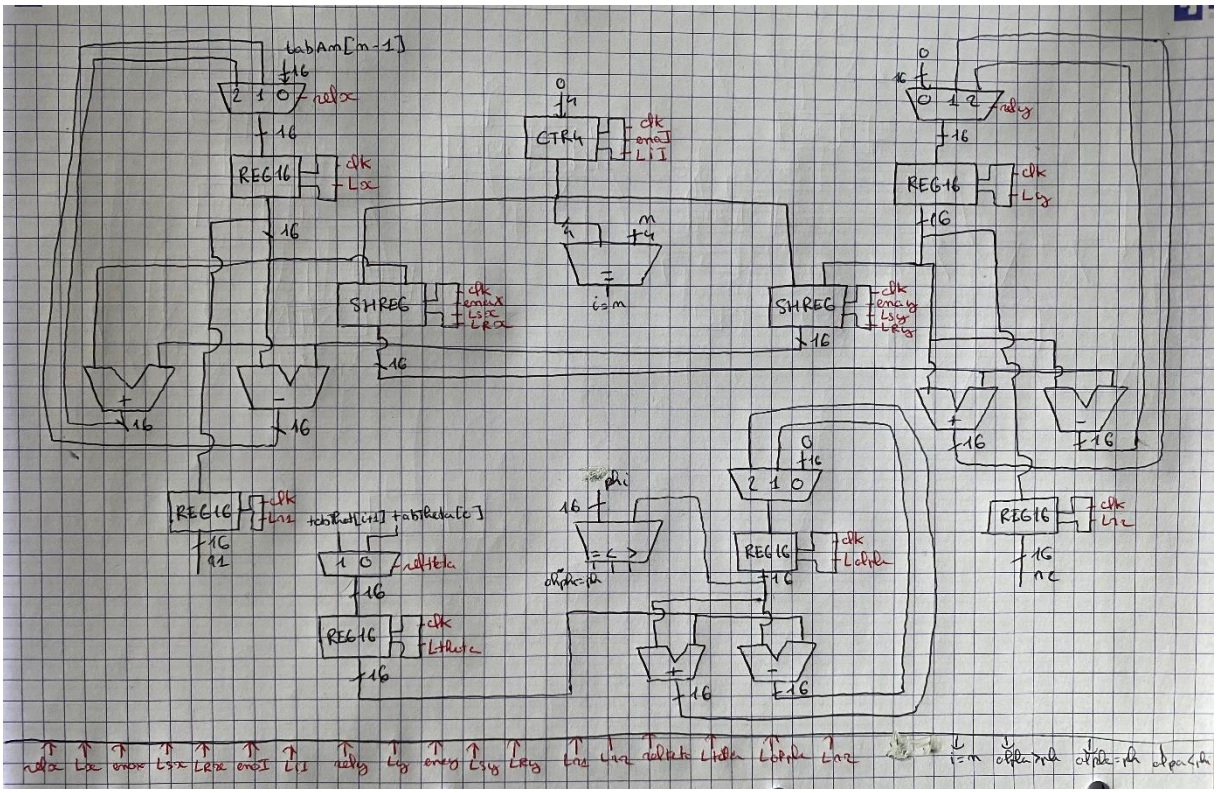
<<Filter>>

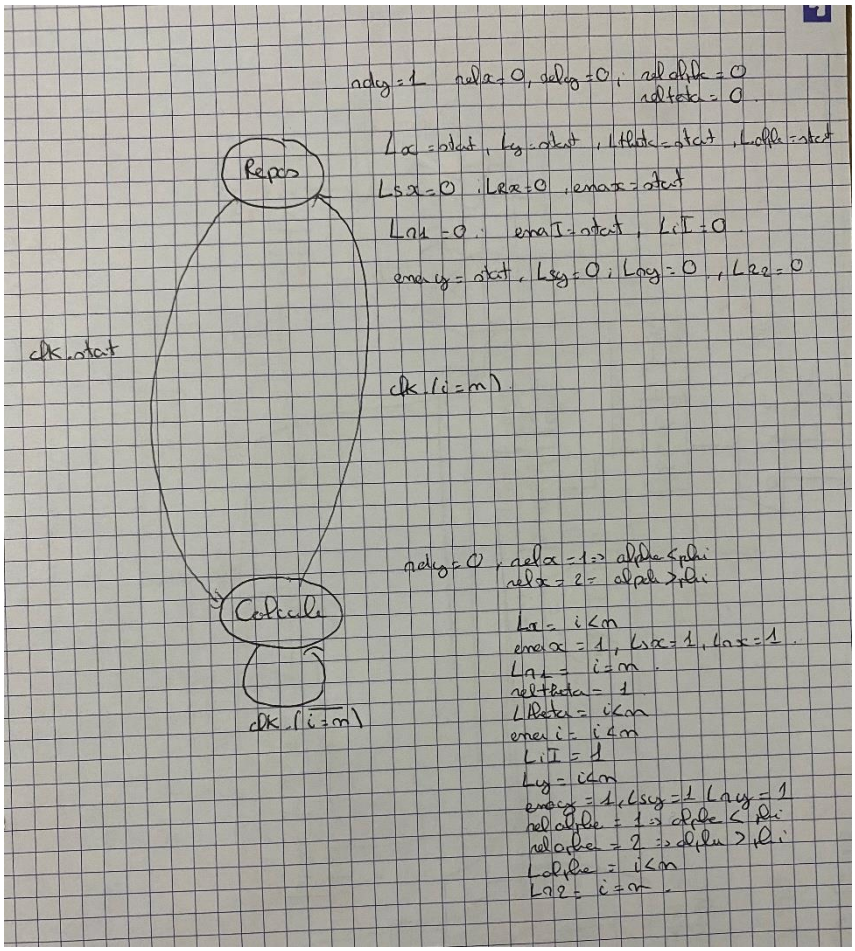
	Fmax	Restricted Fmax	Clock Name	Note
1	160.82 MHz	160.82 MHz	clk	

Nous avons 17 fronts entre le passage de start a 1 et l'affichage du résultat sur r1 et r2. Pour le cas ou Fmax = 176.96Mhz on aura une période de 5.65 ns donc un temps de calcul de 5.65*17 = 96.05ns. Pour le cas ou Fmax = 160.96Mhz on aura une période de 6.21 ns donc un temps de calcul de 6.21*17 = 105.57ns.

II.5- Modèle architectural en VHDL

On obtient à partir du modèle VHDL précèdent la partie opérative + la partie contrôle suivante.





III- Conclusion

Ce TP nous a permis de comprendre les étapes afin d'implémenter un algorithme sur du matérielle. Nous avons compris l'intérêt de partir d'un code C, en virgule flottante de le convertir en virgule fixe afin de simplicité l'implémentation de celui-ci en VHDL. Nous avons aussi vu l'importance de la quantification qui nous a permis de nous apercevoir que dépasser un certain nombre d'itération était inutile. Finalement, en procédant de cette façon, on s'est rendu compte que l'implémentation en VHDL et vraiment une toute petite partie de l'implémentations matérielles.