Module EE7EN202
Projet VHDL : Calculatrice Graphique

LIENARD Simon, **PECORARO** Gabriel
Professeur Encadrant :
BORNAT Yannick

Décembre 2023 - Janvier 2024

Table des matières

1. Cahier des Charges

1.1. Objectifs

Au commencement, nous avons décidé que notre projet consisterait à implémenter une calculatrice graphique sur notre carte. Les objectifs initiaux de notre projet étaient les suivants :

- Réalisation de tracés de fonctions de référence telles que la fonction exponentielle ou le cosinus sur un module d'affichage mis à disposition sur notre carte.
- Possibilité de personnaliser les coefficients de ces fonctions pour une flexibilité accrue.

1.2. Fonctionnement

Le fonctionnement de l'architecture souhaitée est le suivant. Tout d'abord, une remise à zéro globale est nécessaire pour effacer l'affichage sur la calculatrice et la rendre prête à afficher une nouvelle fonction. La fréquence de fonctionnement de l'architecture est de $100MHz$. Enfin, les boutons d'entrée permettent la gestion des coefficients.

1.3. Choix de l'écran

Notre objectif est d'afficher des courbes correspondant à des fonctions que nous avons préalablement choisies. Ainsi, plusieurs options se sont présentées à nous. Le premier choix était l'utilisation d'un écran VGA, tandis que le second était l'utilisation du module Pmod OLED RGB.

Pour faire un choix préliminaire et commencer avec des idées clairement énoncées, nous avons opté pour la solution présentant le plus d'avantages pour notre groupe. Tout d'abord, nous avons relevé l'avantage indiscutable de l'écran VGA, qui est significativement plus grand que celui du PMOD. Ainsi, nous pouvons réaliser des tracés plus précis. Cependant, le fait que cet écran soit plus grand d'un facteur d'échelle supérieur à la dizaine par rapport au PMOD OLED peut également poser des problèmes logistiques dans la réalisation de notre projet. Il est donc plus facile d'utiliser le PMOD chez nous pour réaliser des tests que d'emprunter un grand écran VGA.

Par ailleurs, malgré sa taille plus réduite, le PMOD nous permettrait tout aussi bien d'afficher les courbes de tendance que l'écran VGA. En étudiant les documentations mises à disposition, nous avons découvert que le PMOD OLED disposait déjà d'un bloc nommé Sigplot, facilitant le tracé de courbes sur ce module. En revanche, l'écran VGA ne propose pas de solutions aussi simples pour le tracé de courbes.

1.4. Répartition des tâches

Avant de démarrer notre projet, nous avons convenu de la répartition des tâches parmi les membres du groupe pour clarifier les responsabilités et éviter tout chevauchement d'activités dû à un manque de communication. L'objectif principal était d'optimiser notre temps, et pour ce faire, nous avons établi un plan dès le début du projet.

En premier lieu, nous avons identifié la première difficulté à surmonter : la création d'un module permettant de calculer la fonction exponentielle. Pour cette étape initiale, nous avons choisi de collaborer à deux, favorisant ainsi des discussions approfondies sur les solutions à mettre en œuvre. Cette approche nous a permis de sélectionner la meilleure solution en ayant une perspective croisée. De plus, la première fonction a servi de modèle pour les autres, partageant ainsi une architecture similaire. Cette approche nous a également permis de comprendre en profondeur la réalisation de calculs complexes dans un contexte binaire, travaillant exclusivement avec des bus de bits.

Après avoir achevé la première fonction et vérifié son bon fonctionnement, nous avions prévu que l'un des membres (Gabriel) se concentrerait sur la mise en place des autres fonctions en utilisant la même structure que celle de la fonction exponentielle. Parallèlement, le deuxième membre (Simon) se concentrerait sur l'implémentation de l'affichage, avec pour objectif d'afficher la fonction créée précédemment. Bien que nous ayons anticipé initialement que cette partie du projet serait rapide et simple grâce à l'utilisation d'un module existant, nous avons rencontré des difficultés inattendues, comme discuté dans les sections appropriées. En conséquence, notre disposition initiale a dû être ajustée au cours du projet.

2. La mise en place des calculs

Dans cette partie, nous allons expliquer les démarches entreprient pour réaliser les calculs, les stratégies mathématiques adoptés afin de réaliser nos calculs.

2.1. Le cas de la fonction exponentielle

La première fonction que nous avons décidé de tracer est la fonction exponentielle, car selon nous, elle constituait la meilleure option pour commencer. Cette fonction ne peut pas être tracée de la même manière que les fonctions linéaires ou même paraboliques en utilisant une relation qui relie l'abscisse à l'ordonnée. Cependant, avec la solution que nous avions en tête pour réaliser les tracés, elle était la plus simple.

Commençons par présenter le protagoniste de cette sous-partie. La fonction exponentielle est alors l'unique fonction dérivable solution de ce problème de Cauchy :

$$f' = f$$
$$f(0) = 1$$

Si l'on trace la courbe correspondant à cette fonction, on obtient une courbe semblable à celle-ci :

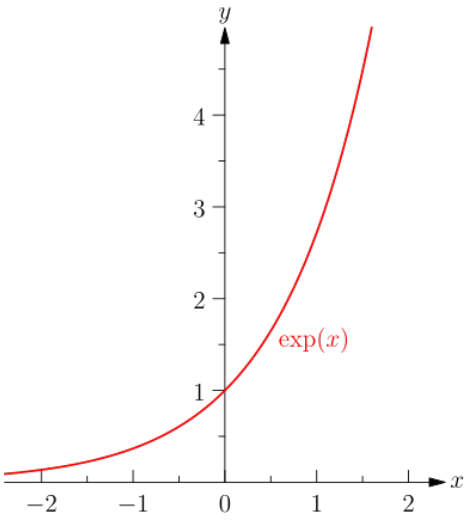

FIGURE 1 – Fonction Exponentielle

Cependant, il devient rapidement évident que ce n'est pas avec les conditions du problème de Cauchy définissant cette fonction que l'on va tracer l'exponentielle. C'est pourquoi nous allons utiliser une autre relation qui définit également cette fonction. En utilisant la notion de série entière étudiée en classe préparatoire, on peut exprimer l'exponentielle sous cette forme :

$$exp(x) = \sum_{n=0}^{+\infty} \frac{x^n}{n!}$$

On remarque qu'en utilisant cette relation, nous pourrons calculer les images pour des abscisses données à l'aide de la fonction exponentielle. Cependant, il est à noter que le calcul ne peut être effectué lorsque n tend vers l'infini. Nous devrons donc tronquer notre série entière. À cette fin, l'utilisation d'un langage de programmation nous permettra de visualiser la forme de notre courbe en fonction du plus grand n choisi :

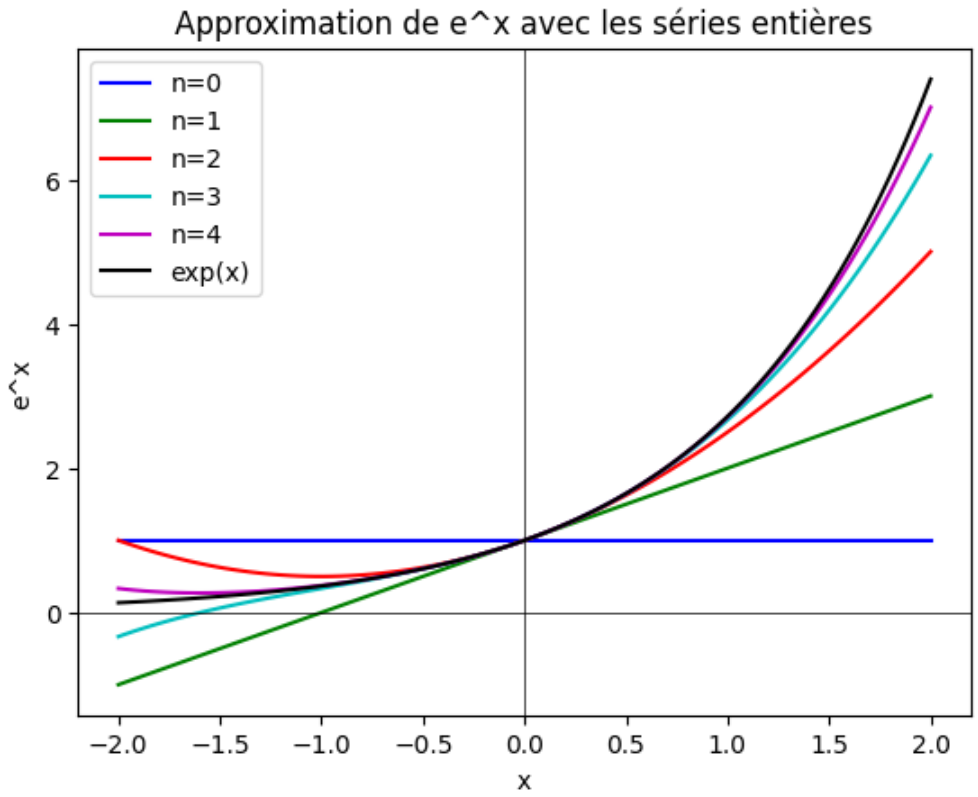

FIGURE 2 – Approximation de la fonction exponentielle en passant par les séries entières

On remarque qu'en prenant $n = 3$, on obtient un bon compromis entre la complexité du calcul et le fait que la courbe se rapproche de celle de l'exponentielle. Ainsi, en prenant un intervalle d'étude entre $[-1; 1[$, les deux courbes sont presque identiques.

Maintenant que l'on a défini notre domaine d'étude et la méthode que l'on va mettre en place pour tracer la fonction, on peut commencer à mettre cela en place en VHDL.

D'abord, on ne peut pas directement travailler avec des nombres non entiers en binaire ni réaliser l'opération de division. Cependant, on travaille dans l'intervalle $[-1; 1]$. On va donc utiliser l'astuce de changer notre échelle. Ainsi, on aura ce changement :

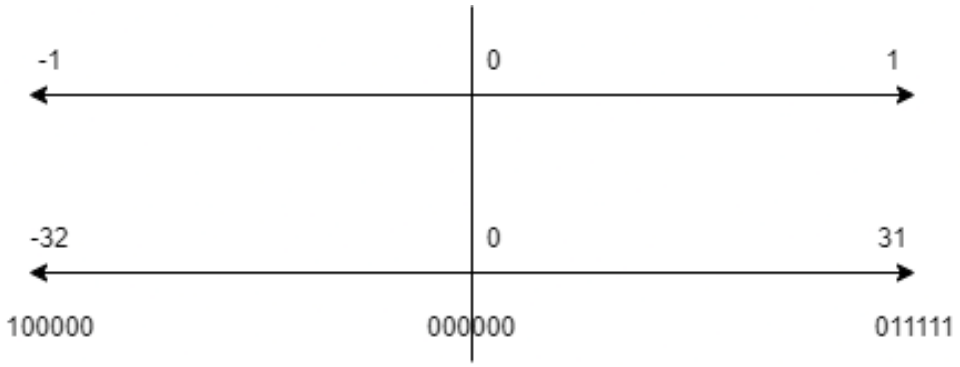

FIGURE 3 – Changement d'échelle

Mais ce changement d'échelle va également nous permettre de réaliser les calculs des séries entières. Car initialement, on devait implémenter ce calcul :

$$exp(x) = 1 + x + \frac{x^2}{2} + \frac{x^3}{6}$$

Mais maintenant que l'on a changé d'échelle et que l'on prend $u_{32} = x$, on va se retrouver avec cette équation :

$$exp(u_{32}) = 32 + u_{32} + \frac{u_{32}^2}{64} + \frac{u_{32}^3}{6144}$$

Pour mettre en place ces calculs, on va alors utiliser quelques propriétés d'arithmétique. Nous allons juste réaliser un lien entre la base 10 et la base 2 :

En base 10 :

$$\frac{2024}{10} = 202,4 = \frac{202 * 10 + 4}{10}$$

On se retrouve alors à déplacer à droite la virgule quand on divise par 10 et le chiffre qui passe derrière la virgule devient le reste.

Et en base 2, on a la même chose :

$$\frac{1000101_{base2}}{10_{base2}} = 010010_{base2} + 0.1_{base2}$$

Cependant, notre travail s'effectue sur un écran de quelques centimètres de dimension avec une résolution légèrement inférieure à 100 pixels. Dans ce contexte, la préservation du bit restant pour obtenir la précision de la courbe est négligeable. Par conséquent, nous avons décidé de le négliger.

On va alors mettre en place la stratégie décrite sur le schéma ci-dessous afin de réaliser les calculs :

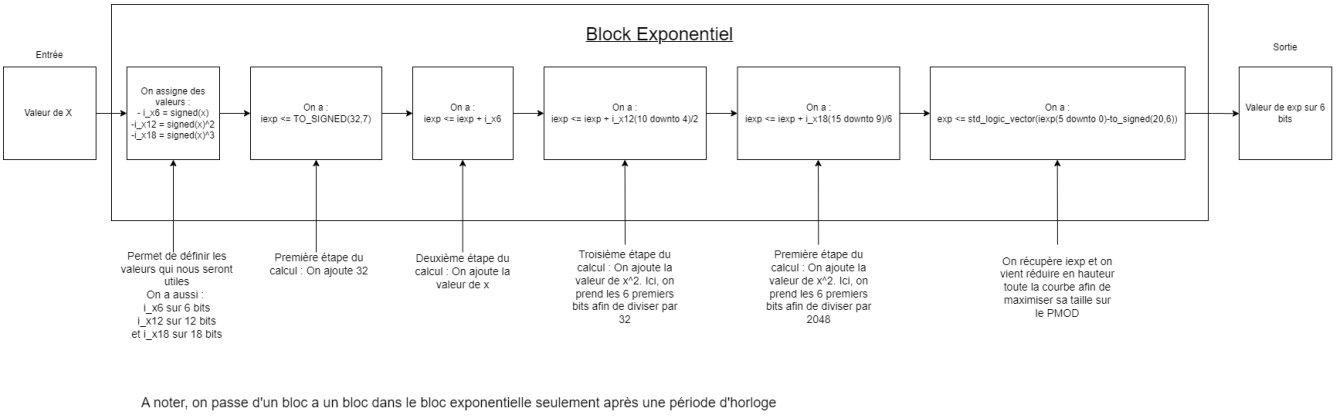

FIGURE 4 – Diagramme Bloc exp

Avec ce fonctionnement d'implémentation et en réalisant une architecture nous permettant d'obtenir la courbe en entier, on obtient cette courbe :

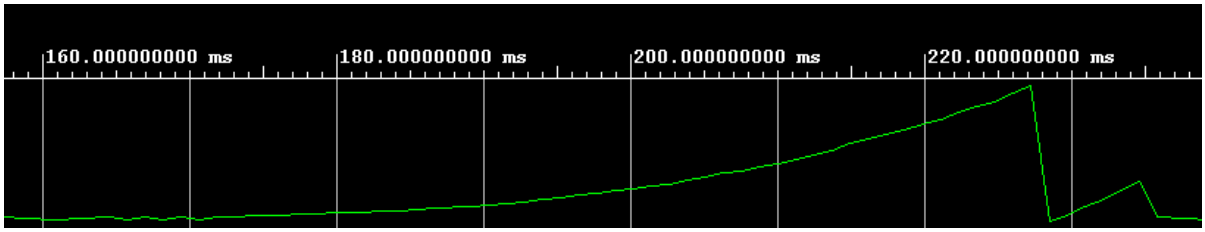

FIGURE 5 – Tracé de la fonction exponentielle

Finalement, on remarquera qu'on observe un débordement à la fin de notre courbe. On peut alors le retirer en réduisant le nombre de points affichés par notre PMOD en l'arrêtant juste avant.

2.2. Autres fonctions

Dans cette partie, nous allons brièvement expliquer comment nous avons généré l'affichage des autres fonctions. Nous allons aborder dans un premier temps la mise en place de fonctions simples telles que des paraboles ou des fonctions linéaires. Enfin, dans un second temps, nous allons discuter de la mise en place de fonctions plus complexes telles que des applications sinusoïdales ou non linéaires.

2.2.1 Fonction linéaire et parabole

Pour tracer les fonctions telles qu'une fonction affine ou la valeur absolue, cela n'a pas posé de problème majeur. La fonction affine est un simple compteur sur des nombres signés allant de $[-32, 31]$ avec un pas de 1. Lorsque l'on arrive à 31, on reboucle sur -32, de même lorsque le reset est activé.

On a appliqué la même méthode pour la fonction valeur absolue, qui fonctionne sur le même principe que la fonction affine. La seule variante est la disjonction de cas effectuée pour rendre positifs les nombres entre $[-32, -1]$.

Pour la fonction carré, on a mis en place un algorithme qui nécessite 5 coups d'horloge pour le calcul. L'idée est que l'on crée une variable intermédiaire qui transforme le bus de bits en sortie du compteur sur $[-32, 31]$ en nombre signé. On charge dans une nouvelle valeur le carré de la valeur signée du compteur. On tronque cette valeur sur 6 bits pour faire disparaître les bits de signe et éviter les problèmes de débordement.

2.2.2 Fonctions non linéaires

L'enjeu majeur du projet était de proposer également des fonctions un peu moins évidentes à tracer, telles que des bijections, des sinusoïdes ou des fonctions non linéaires. Après avoir réussi à tracer l'exponentielle, nous avons essayé de transposer l'algorithme de ce dernier sur les autres fonctions. Bien que cela s'appliquait plutôt bien à des fonctions telles que arcsin ou arctan, nous avons rencontré de nombreuses difficultés à généraliser cette méthode.

En effet, si on prend en compte la fonction cosinus, on aimerait afficher sur l'écran une sinusoïde parfaite et mettre en relief les périodes. Or, les séries entières pour les fonctions sinusoïdales ne sont qu'une approximation polynomiale au voisinage de zéro, ce qui biaise totalement les calculs. On voit donc que le traçage de fonctions avec les séries entières en utilisant la virgule flottante a une limite. On illustre ce problème ci-dessous :

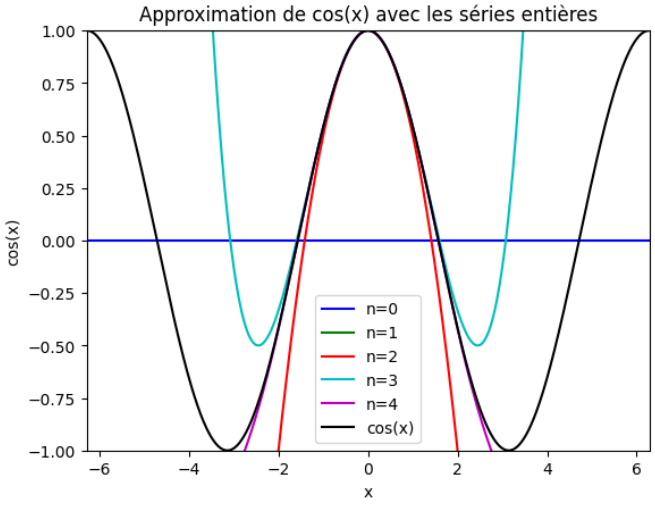

FIGURE 6 – Cosinus et Série Entière avec nombre fini d'itérations

On remarque que pour un nombre fini d'itérations, le polynôme approximant le cosinus n'a plus une forme sinusoïdale, et ce assez rapidement. De fait, pour résoudre ce problème, nous avons préféré tracer un cosinus point par point, ce qui nous garantit beaucoup plus de précision.

De plus, lors du tracé de fonctions non linéaires, nous avons également rencontré des problèmes avec les débordements. En effet, lorsque l'on utilise les séries entières, notamment pour le logarithme ou la fonction réciproque du sinus, en raison de leurs variations, on atteint très rapidement le débordement sur l'intervalle de tracé. C'est encore une fois une illustration des limites algorithmiques du tracé en utilisant les séries entières. Pour résoudre ces problèmes, on a essayé sur la fonction logarithme de contrôler la série entière, en imposant des variations plus faibles en fonction de l'endroit où l'on se situe sur l'intervalle, mais en vain. Pour arcsin, le problème se posait surtout aux bornes car c'est là où arcsin croît le plus rapidement. On a donc décidé d'arrondir manuellement la courbe d'arcsin pour obtenir la forme souhaitée.

3. Discussions

Dans cette partie, nous allons prendre du recul sur l'approche que nous avons eue en ce qui concerne le tracé des fonctions, et nous discuterons également de la mise en place de coefficients entiers sur nos fonctions. Enfin, nous conclurons cette partie sur d'autres approches algorithmiques que nous aurions pu appliquer sur le tracé.

3.1. La méthode de tracé

Comme expliqué, la méthode de la série entière présente de nombreux avantages, notamment sa simplicité qui se résume globalement à une somme, mais aussi le fait qu'elle utilise peu de ressources (cf). Ainsi, cette méthode est simple à mettre en place en pratique, ce qui lui a valu d'être sélectionnée. Néanmoins, comme on l'a expliqué précédemment, on a peu d'influence sur l'évolution du polynôme généré par cette somme, et donc il est très facile d'aboutir à des problèmes de débordement. Nous avons décidé de mettre en place cette méthode aussi grâce à sa flexibilité. Étant donné que l'expression d'une série entière est générale pour tous les éléments réels et complexes, elle nous facilitait la mise en place des coefficients, car les calculs ne sont pas pré-enregistrés dans une RAM, mais sont recalculés à chaque remise à zéro et à chaque coup d'horloge.

3.2. Mise en place des coefficients

3.2.1 Des Difficultés

Nous avions prévu de mettre en place un jeu de coefficients pour chaque fonction, ce qui n'a pas pu être effectué. Tout d'abord, nous voulions être sûrs de maîtriser les fonctions dans leur intégralité, mais aussi d'en proposer le plus possible. Nous nous sommes donc attardés pendant plusieurs séances sur la mise en place des calculs. De plus, comme expliqué précédemment, en raison des nombreux problèmes de débordement que nous n'avions pas réussi à résoudre intégralement, nous n'avons pas eu le temps de réfléchir à la mise en place de ces coefficients, ce qui aurait généré encore plus d'erreurs. Enfin, pour certaines fonctions, nous avons tracé sur certains segments de notre intervalle d'affichage des points à la main. À partir du moment où nous nous sommes résignés à faire cela, la création de ce jeu de coefficients est devenue compromise. De plus, nous avons pensé que mettre des coefficients sur un écran aussi petit, où le maximum est en haut de l'écran, n'était pas une bonne chose pour l'affichage.

3.2.2 Mise en Place Théorique

Néanmoins, bien que nous n'ayons pas pu mettre en place cette variation de coefficients, nous avons quand même réfléchi à comment nous aurions pu le faire. Tout d'abord, le jeu de coefficients aurait été programmé sur les boutons de la carte FPGA. Respectivement, les boutons BTNL et BTNR auraient servi pour incrémenter et décrémenter la valeur des coefficients, BTND et BTNU auraient servi à naviguer entre les différents coefficients. On avait pensé à mettre 3 coefficients. Ce nombre a été choisi par rapport à la fonction parabole qui présentait le maximum de coefficients programmables, soit 3. Pour gérer l'incrémentation et la décrémentation, ainsi que la navigation entre les coefficients, on aurait programmé une FSM avec des états variables qui aurait permis de changer entre chaque état. De plus, on aurait ajouté à cela sur l'afficheur 7 segments la valeur du coefficient actuel.

3.2.3 Algorithme de Cordic et résolution de l'équation de bijection

Avec tous les problèmes que nous avons rencontrés, on peut se demander pourquoi nous n'avons pas opté pour d'autres algorithmes. Tout d'abord, comme expliqué précédemment, nous pensions qu'un algorithme basé sur les séries entières était la meilleure solution, et nous ne nous sommes pas documentés sur d'autres possibilités au vu de l'envergure du projet. Néanmoins, nous avons appris en cours de projet qu'il existait d'autres possibilités. L'idée dans cette partie serait de montrer comment nous aurions pu mettre en place ces algorithmes et en quoi ils auraient pu nous avantager.

La première option aurait pu être l'utilisation de l'algorithme de Cordic. C'est un algorithme couramment utilisé dans les FPGA pour tracer des fonctions non linéaires. Il nécessite très peu de ressources, étant donné qu'il présente une architecture en pipeline. En somme, le plus gros avantage de l'algorithme de Cordic est sa structure simple qui n'utilise que des additions et des décalages de bits. Cela aurait permis d'éviter l'architecture complexe générée par les séries de Taylor. Néanmoins, la vraie complication aurait été l'approche théorique qu'il y aurait eu derrière, et en termes de temps, ce n'était pas faisable pour nous.

La seconde option, pour tracer notamment des algorithmes de fonction réciproque, aurait été de se ramener à la définition de la fonction bijective. Considérons une fonction f bijective de I dans F, on devrait résoudre ceci :

$$f : I \to F$$

$$\forall x \in I, \forall y \in F$$

$$y = f(x)$$

Si cette équation admet une solution, on a la fonction réciproque de F dans I :

$$f^{-1} : F \to I$$

Où f^{-1} est le résultat de l'équation pour y. De fait, on aurait pu, pour chaque valeur d'une fonction bijective, calculer sa fonction réciproque. On aurait procédé par méthode dichotomique pour obtenir l'image réciproque. Cela aurait constitué un algorithme comprenant des sommes avec un pas constant. Encore une fois, par manque de temps, nous n'avons pas pu nous pencher sur une potentielle mise en œuvre de cette méthode.

4. Module d'affichage

Comme expliqué précédemment, nous avons décidé pour notre projet d'utiliser l'afficheur Pmod OLED. Dans cette partie, nous allons expliquer comment nous avons intégré cet élément dans le fonctionnement de notre calculatrice.

4.1. Mise en place du module

Avant de commencer à décrire l'architecture des blocs permettant d'afficher nos courbes, nous avons commencé par étudier et comprendre la datasheet du composant.

D'abord, la datasheet nous présente les modules génériques qui nous permettent de personnaliser la façon dont nous allons utiliser notre composant. Ici, le premier module générique est le choix de l'activation ou non du PARAM BUFF. Afin de minimiser la consommation de notre carte en utilisation, nous avons décidé de l'activer et donc d'adapter notre architecture à ce choix. Le deuxième choix était d'activer ou non le générique LEFT SIDE. Ici, étant donné que l'on utilise le module sur la partie droite de notre carte, nous n'avons pas eu besoin d'activer ce paramètre.

Ensuite, nous nous sommes tournés vers l'étude des entrées du module Sigplot. Ainsi, hormis les entrées *clk* et *rst*, les entrées que nous devons mettre en place sont respectivement :

1. $dispshift$
2. *sampleen*
3. *samplenum*
4. *sample*
5. *background*

D'abord, nous avons les entrées $sample - en$ et *background*, qui sont de notre point de vue simplement des entrées esthétiques. Donc, nous n'allons pas nous attarder sur leur fonctionnement, même si nous allons les utiliser pour peaufiner notre projet.

En revanche, les entrées qui vont nous intéresser sont les 3 autres. Il va de soi que nous allons devoir utiliser le bus nommé *sample* afin d'envoyer la valeur à afficher. Aussi, à chaque fois que nous réalisons un calcul, nous allons incrémenter la valeur que nous avons en abscisse. C'est pourquoi nous allons devoir translater d'un carreau vers la droite pour positionner le pixel suivant. C'est alors le rôle de $dispshift$ de réaliser cette translation. Puis, pour valider nos entrées, nous devons à chaque fois activer l'entrée *sampleenable*.

Mais comme nous avons activé le générique buffer, nous devons respecter ce schéma d'envoi de données :

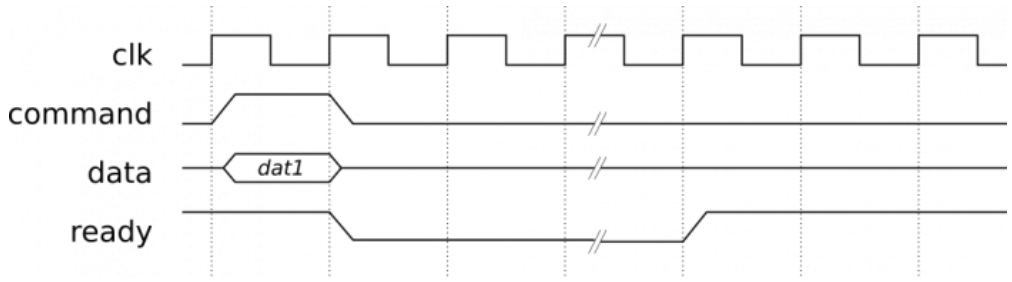

FIGURE 7 – Diagramme time avec PARAM BUFF activé

Image récupérée sur la datasheet mise à disposition sur le site : yannick-bornat.enseirb-matmeca.fr

Donc, on remarque que l'on doit envoyer nos données pendant une période d'horloge après que la sortie *ready* se soit activée. On se rend compte que dans notre implémentation, nous allons devoir alors réaliser un bloc nous permettant de répondre aux demandes du bloc Sigplot en envoyant les données au bon moment. Une architecture sous cette forme va alors être mise en place pour réaliser cette tâche :

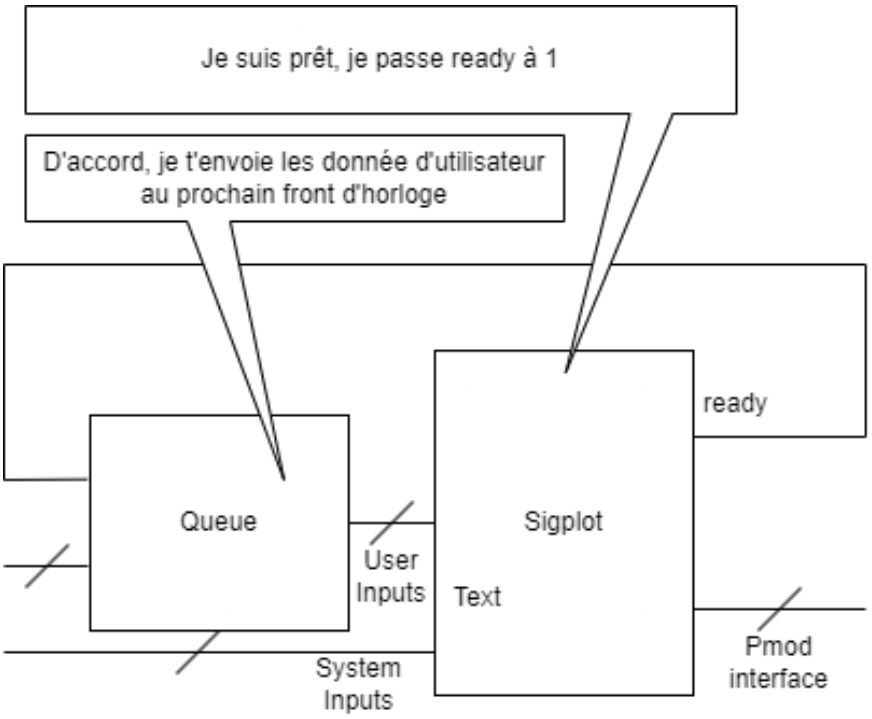

FIGURE 8 – Structure commande de sortie

On peut remarquer que nous avons créé une variable "inter" dans notre bloc, qui permet d'énumérer le nombre de valeurs que l'on envoie à notre PMOD. Cela évite que ce dernier ne s'arrête une fois que toute la fonction a été dessinée.

De plus, on ne sait jamais quand le signal "ready" va s'activer. C'est pourquoi nous allons utiliser une sortie du bloc "queue" pour indiquer quand nous devons incrémenter la valeur de x afin d'effectuer les prochains calculs. Ainsi, lors de l'envoi de nos valeurs à *sigplot*, nous signalons à notre bloc qu'il peut procéder au calcul pour préparer la prochaine valeur à envoyer à *sigplot*. Cependant, on peut considérer que les calculs nécessitent un certain nombre de périodes d'horloge pour être réalisés. Cela n'est cependant pas un problème majeur, car le temps de calcul est négligeable par rapport au temps que *sigplot* met pour traiter les données et donc nous avertir qu'il est prêt.

Une fois tout cela mis en place, on a alors réalisé les simulations pour l'exponentielle seulement et observé que notre description était bien fonctionnelle.

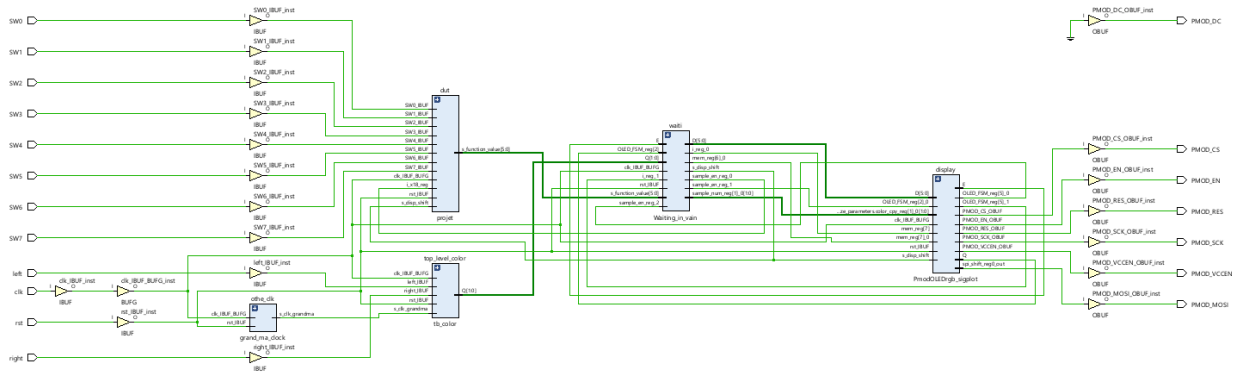

FIGURE 9 – Schematic global du montage

4.2. La réalité du terrain

Après avoir vérifié que nos descriptions étaient correctement fonctionnelles dans la simulation, nous avons généré le bitstream complet et effectué l'implémentation sur notre carte. À notre grande surprise, nous avons constaté que les résultats obtenus différaient considérablement de ceux que nous attendions. Nous ne comprenions pas du tout pourquoi nous obtenions une couleur de courbe aléatoire, même non choisie, à chaque fois, sur une partie aléatoire de notre carte.

Cependant, étant donné que l'affichage constituait la pierre angulaire de notre projet, il était impératif de faire fonctionner cette partie. Dans un premier temps, nous avons tenté de comprendre les sorties que nous obtenions en ajustant les paramètres d'entrée de manière empirique pour observer toute modification de la sortie. Cependant, rien ne changeait.

Ensuite, nous avons remis en question notre décision d'activer le $PARAM\_BUFF$ et avons décidé de le désactiver, en redéfinissant également notre bloc *Queue* pour le faire correspondre à la nouvelle méthode de travail. Malgré une nouvelle simulation semblant confirmer que notre carte devrait fonctionner, nous avons implémenté la description sur notre carte, mais l'écran refusait toujours de nous renvoyer le résultat attendu.

Nous avons alors consacré deux semaines à tenter en vain de modifier de petites parties afin d'observer les changements sur l'écran. Ensuite, nous nous sommes fixés pour défi de contrôler au moins la couleur de la courbe que nous dessinions. Après de nombreux tests, nous avons réalisé que nous ne devions pas modifier la valeur de la courbe, contrairement à ce qui était indiqué dans la datasheet. Nous avons donc décidé d'adopter un fonctionnement similaire à celui indiqué sur ce schéma, en conservant à chaque fois les valeurs même lorsque "ready" se désactivait.

Une fois cette étape achevée, nous sommes entrés dans la phase où nous devions simplement améliorer notre projet en corrigeant quelques problèmes et en ajoutant même des fonctionnalités.

5. Synthèse du circuit

Dans cette partie, nous proposons une brève vue globale de notre circuit, les ressources nécessaires, la consommation, et le schéma global. Après avoir totalement finalisé notre circuit, nous pouvons exhiber le schéma complet du circuit suivant : À première vue, il semble que ce qui nécessite le plus de ressources soit le calcul des fonctions et l'affichage graphique, ce qui semble cohérent. Nous avons également effectué une étude sur le nombre de LUTs, de RAM utilisée, etc. Sur Vivado, on peut également générer un diagramme énergétique. Nous nous proposons donc de les commenter :

Resource	Utilization	Available	Utilization %
LUT	600	63400	0.95
LUTRAM	6	19000	0.03
FF	370	126800	0.29
DSP	5	240	2.08
IO	21	210	10.00
BUFG	1	32	3.13

FIGURE 10 – Tableau des ressources utilisés

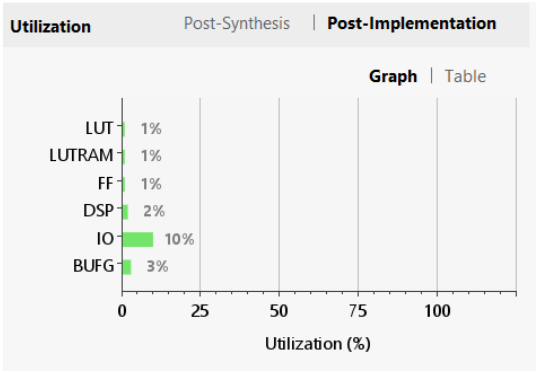

FIGURE 11 – Table Énergétique

On remarque que l'on utilise 600 LUTs et 370 Flip-flops, ce qui est raisonnable pour la taille du projet. Néanmoins, ces deux composants, bien qu'ils soient les plus nombreux, ne consomment pas le plus d'énergie. Ce qui consomme le plus d'énergie sont les entrées et les sorties entre la carte et le monde extérieur, et l'on peut présumer que le PMOD constitue une grande partie de cette consommation. Le buffer présent dans le PMOD est la deuxième plus grande source d'énergie. On peut également noter que nous avons peu de LUTRAM, qui peut être assimilé à une RAM. Cela montre que régénérer à chaque fois les adresses qui codent les fonctions nous a permis de rendre un programme plus optimisé en énergie et en espace mémoire. On remarque que l'on a aussi des blocs DSP. Ces éléments sont spécialisés pour les opérations de traitement de signaux numériques ou d'image. Ces blocs consomment une grande partie de notre énergie.

En somme, on remarque énergétiquement parlant que le PMOD consomme beaucoup sur notre carte. Néanmoins, les valeurs obtenues ne sont pas aberrantes. On va regarder brièvement les performances en termes de puissance statique et puissance dynamique. On a le diagramme suivant :

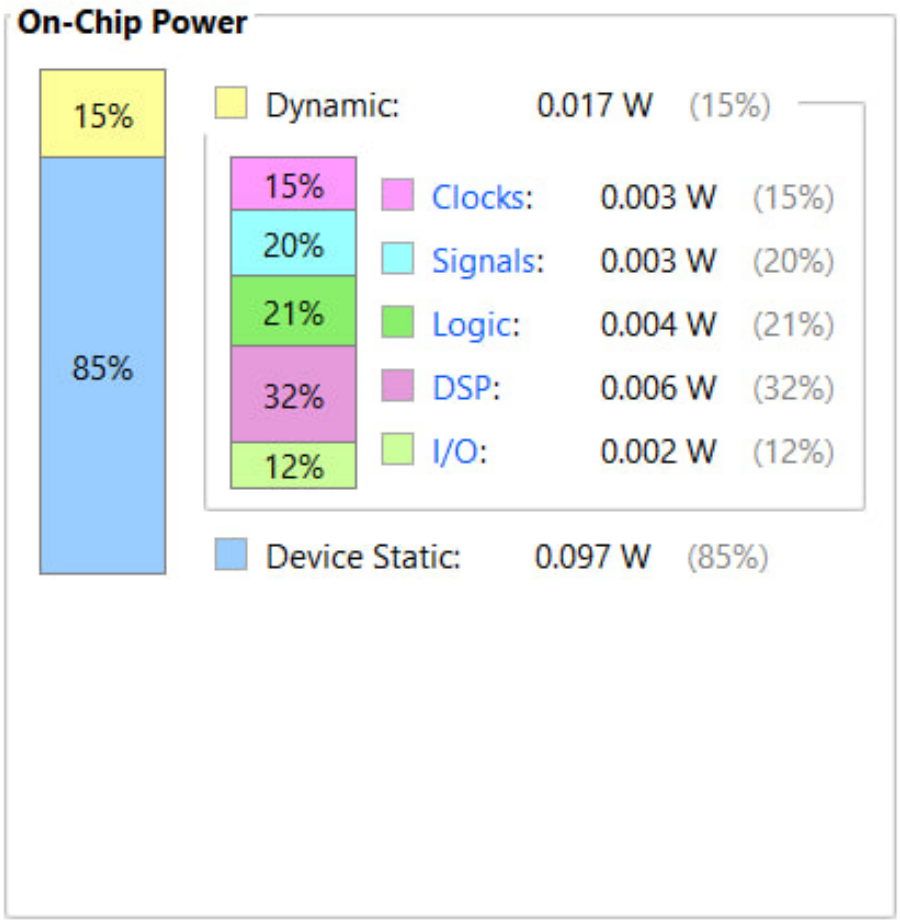

FIGURE 12 – Tableau de Puissance Statique et Dynamique

On remarque avec cette table que dans notre puce, la majeure partie de la puissance est statique. Elle est due aux courants statiques, de fuite et au phénomène d'effets tunnels liés à la mémoire qui se passent dans la puce. En ce qui concerne la puissance dynamique, qui est due aux commutations, la majeure partie vient du traitement du signal numérique dans les DSP. Alors que les entrées et sorties représentaient la plus grande puissance en post-implémentation, ici c'est la plus faible, ce qui montre encore une fois que les entrées et sorties du PMOD tirent le plus d'énergie. Enfin, les signaux logiques et l'horloge constituent aussi une grande partie de la puissance statique.

6. Conclusion

Pour conclure sur l'ensemble du projet, ce fut vraiment une belle expérience de concevoir du début à la fin son propre projet FPGA. Cela nous a permis de mettre en place tout ce que nous avons appris tout au long de notre formation, mais aussi de découvrir de nouveaux composants. De plus, nous avons vraiment pu nous immerger dans le travail d'un ingénieur en surmontant des épreuves, en résolvant ou contournant des problèmes pour arriver à nos fins. C'est une excellente initiative de mettre en place ces projets.

Remerciements à Monsieur Yannick Bornat qui nous a aidés et supervisés tout au long du projet, et qui nous a permis d'accomplir une partie de notre dessein.

7. Sitographie

1. https ://fr.wikipedia.org/wiki/Fonction_exponentielle
2. yannick-bornat.enseirb-matmeca.fr/wiki/doku.php ?id=en202 :pmodoledrgb
3. An FPGA Implementation of the Natural Logarithm Based on CORDIC Algorithm, Shaowei Wang, Yuanyuan Shang, Hui Ding, Chen Wang and Junming Hu College of Information Engineering, Capital Normal University, Beijing 100048, China
4. A fixed-point implementation of the natural logarithm based on a expanded hyperbolic CORDIC algorithm ,Daniel R. Llamocca-Obregón, Carla P. Agurto-Ríos