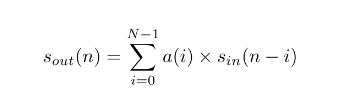
**Projet de Conception d'Électronique Numérique Intégrée**

**Réalisation d’un filtre passe-bas**

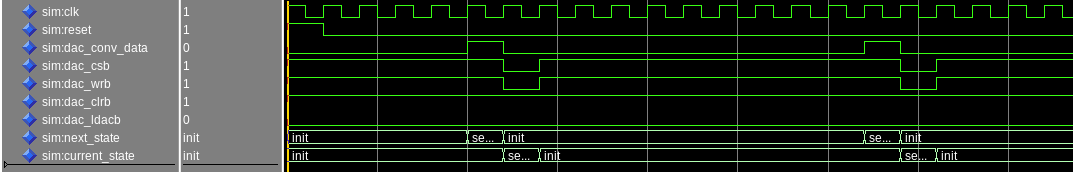
Dans cette série de TP, nous avons étudié et conçu un filtre passe bas à l’aide du langage de description VHDL.

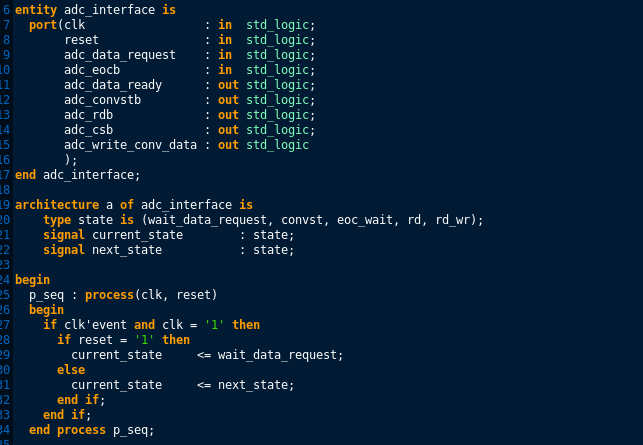
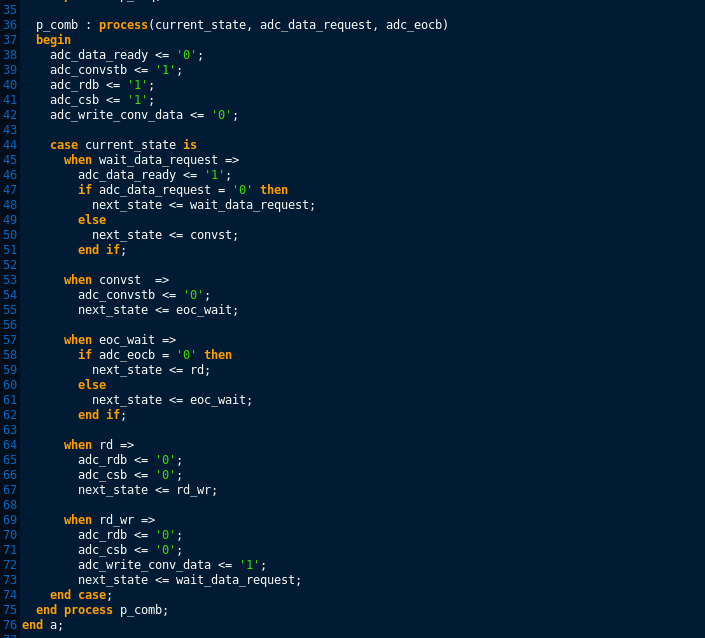
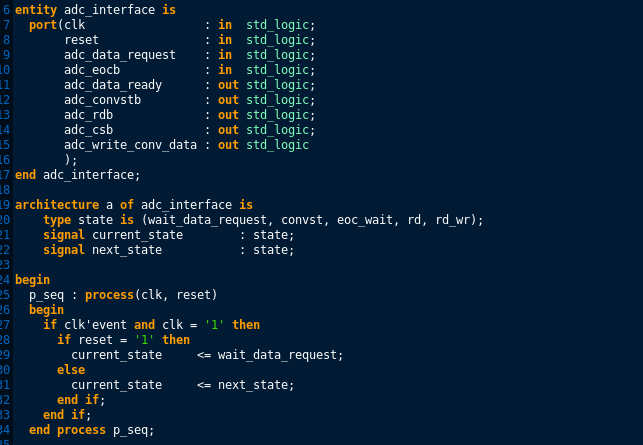
**1** - Nous avons compris le fonctionnement de chaque partie de l’architecture du filtre. Pour comprendre, on doit utiliser les figures 6, 7 et 8 du sujet, ainsi que cette formule :

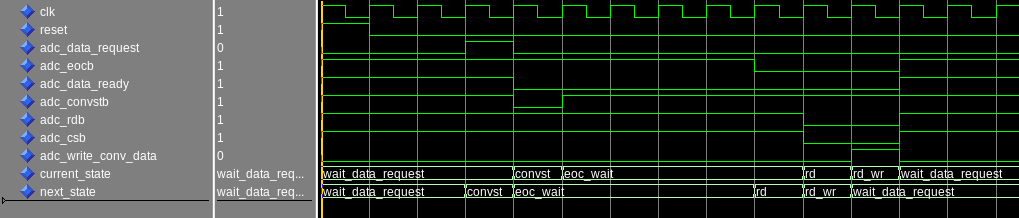


En regardant la figure 7, on comprend que que le système ne contient qu’un seul additionneur et qu’un seul multiplieur dans le but d’éviter des coûts inutiles.

En regardant la figure 8 et le fichier rom.vhd, on comprend que la FSM s’occupe de compter de 0 à 31 et qu'en sortie, la partie RAM coeff renvoie une valeur sur 8 bits en fonction de la valeur d’entrée.

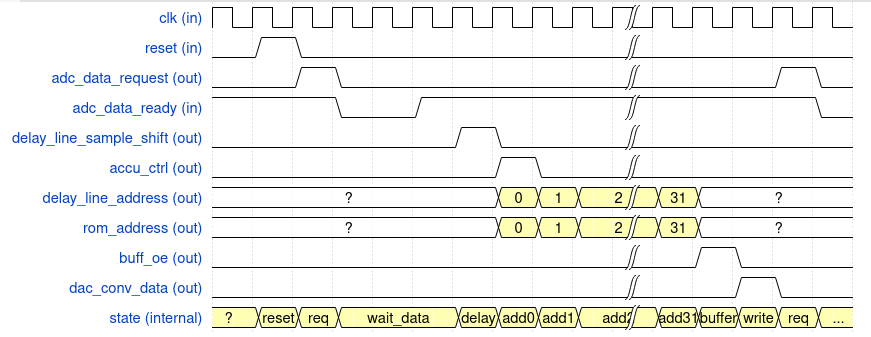
**2** - Pour le bloc DAC, nous n’avons pas écrit le fichier dac\_interface.vhd car ce dernier était déjà écrit, mais nous avons tracé le chronogramme de la simulation. 

Nous avons en revanche dû écrire le fichier adc\_interface.vhd :

Dans le bloc ADC, il y avait 5 états possibles. Nous avons également tracé le chronogramme de la simulation, qui montre que notre ADC a le bon comportement.

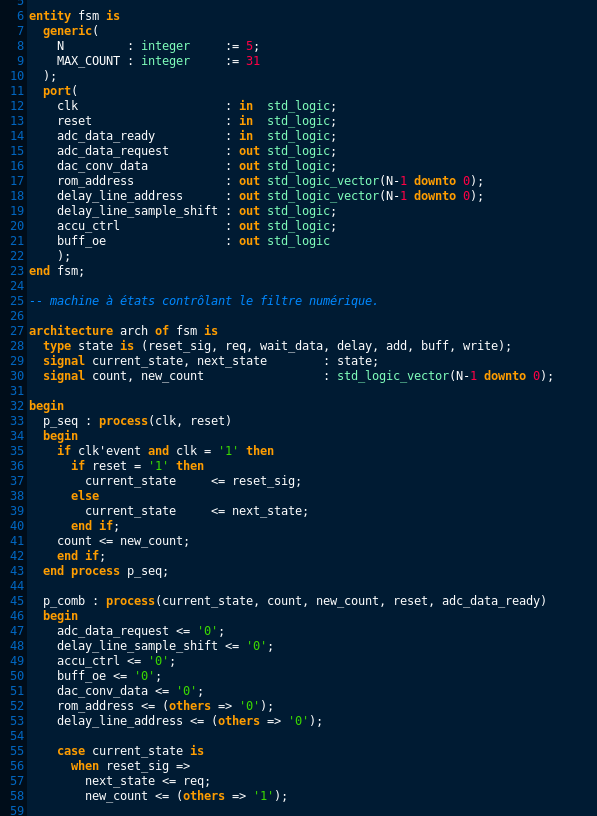
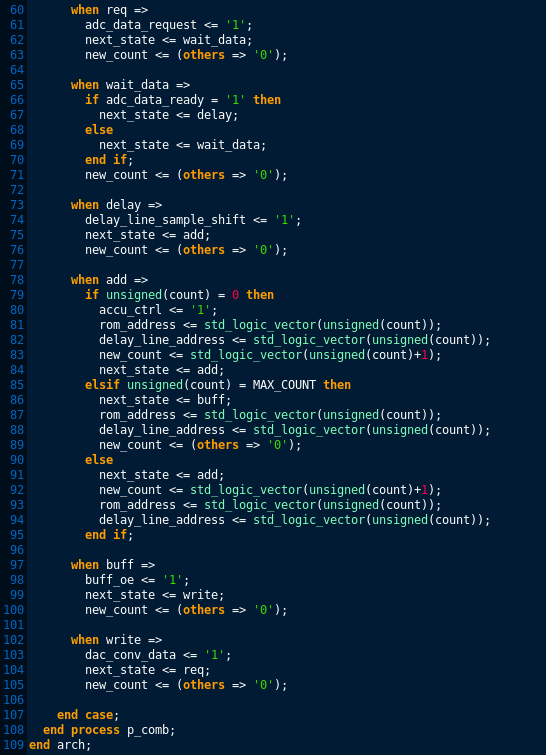
**3.a & 3.b** -

Après avoir longuement étudié les figures 8 et 9, voici le chronogramme théorique représentant la fsm que nous avons fait :

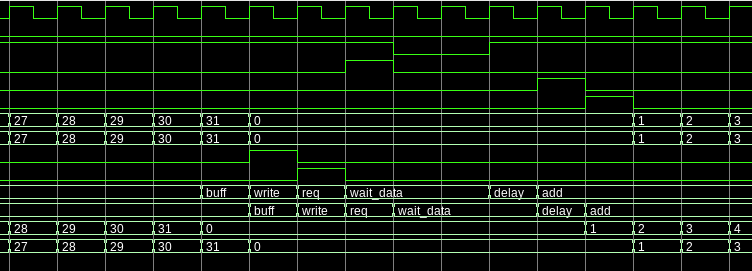
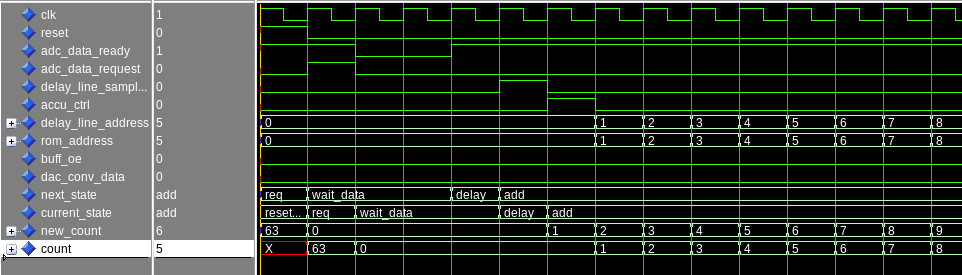
Nous avons créé différents états permettant d’implémenter le comportement voulu sous forme de machine à état. Les états sont les suivants : *reset*, *req*, *wait\_data*, *delay*, *add0* … *add31*, *buffer* et *write*.

**3.c** -

Dans notre implémentation en VHDL de la FSM, nous n’avons pas voulu faire 31 états *add* différents (*add0*, *add1*, … , *add31*) comme indiqué dans le chronogramme ci-dessus. Nous avons donc choisi de faire un seul état add avec un compteur sur 5 bits que l’on incrémente. L’état add se termine lorsque le compteur atteint la valeur décimale 31 (ce qui représente 32 passages). Voici notre implémentation de la FSM :



Nous avons ensuite écrit un test\_bench permettant de simuler le comportement du chronogramme théorique ci-dessus. Le chronogramme que nous avons obtenu à partir de la simulation est donné ci dessous :

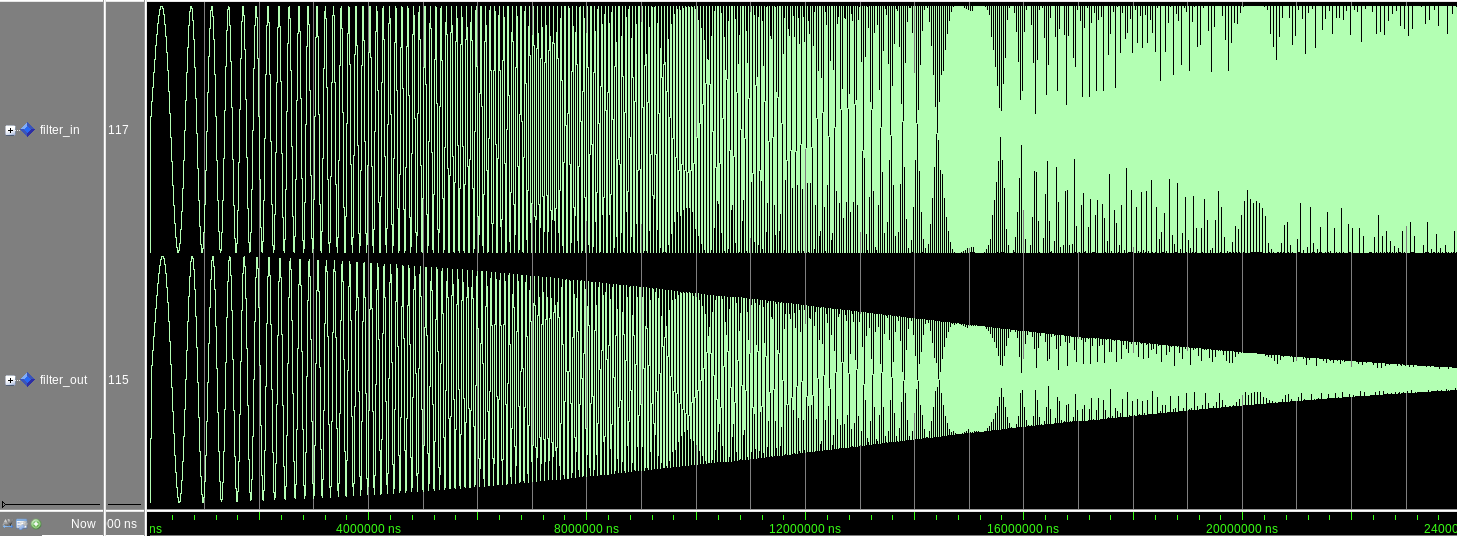


On observe bien le comportement décrit dans le chronogramme que nous avions tracé de manière théorique. Le seul point particulier de ce chronogramme est la valeur 63 prise par *count* et *new\_count* au début de la simulation. Cette valeur s’explique par une erreur de notre part. En effet, on peut voir dans le code que dans l’état *reset* (ligne 55), nous mettons tous les bits de *new\_count* à 1. Or au début, nous avions mis N=6, ce qui fait que la valeur max était 63 et non 31.

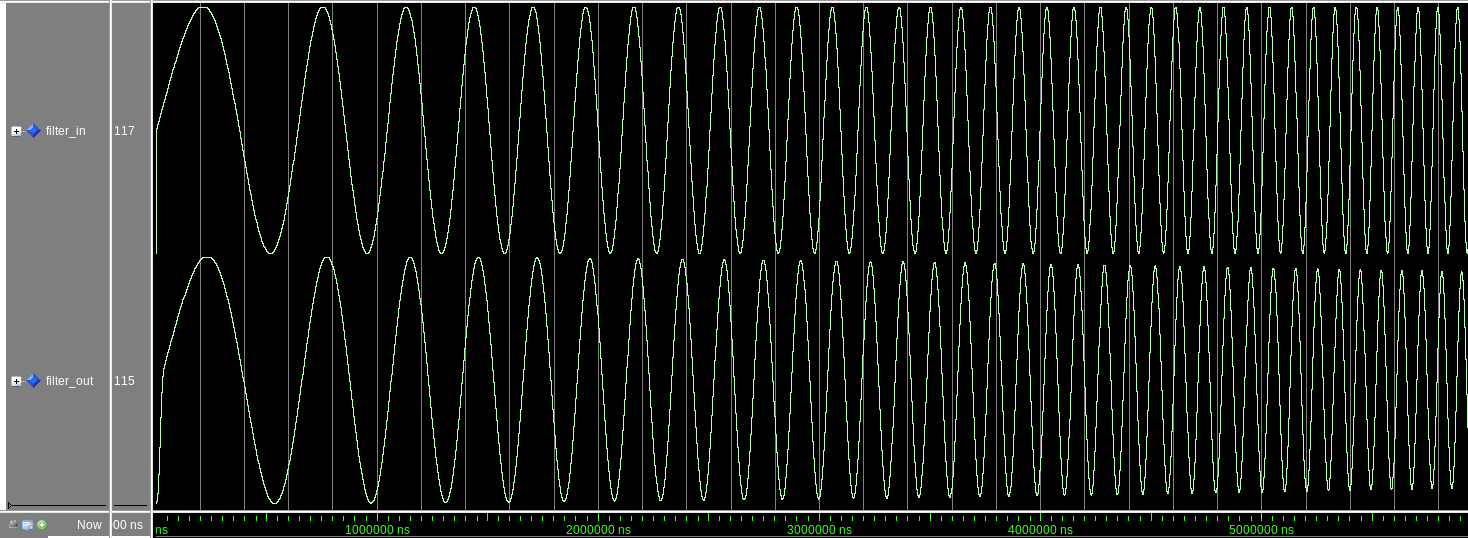
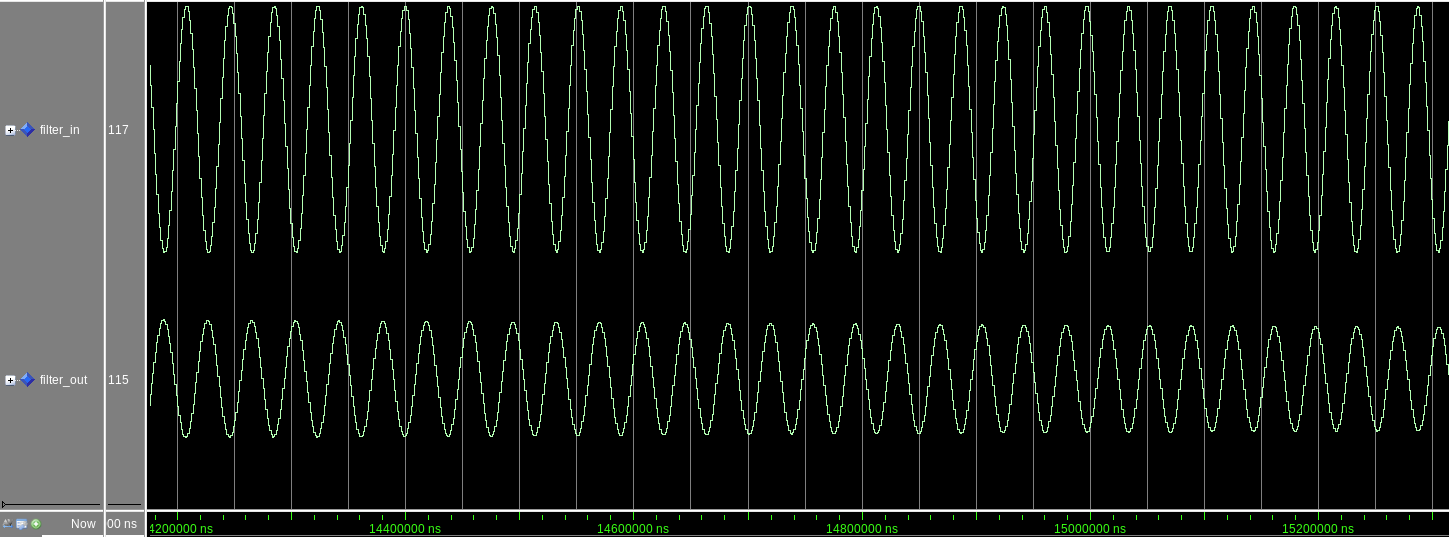
La capture d’écran du code a été faite après que nous ayons corrigé cette erreur, alors que la capture d’écran du chronogramme a été faite avant la correction, d’où l’erreur visible sur le chronogramme. Mais cela ne change pas le fait que la suite d’état est la bonne et que la fsm se comporte comme attendue. Vous pourrez d’ailleurs voir que dans les captures du chronogramme après l’optimisation, *new\_count* vaut bien 31 après après un *reset*.

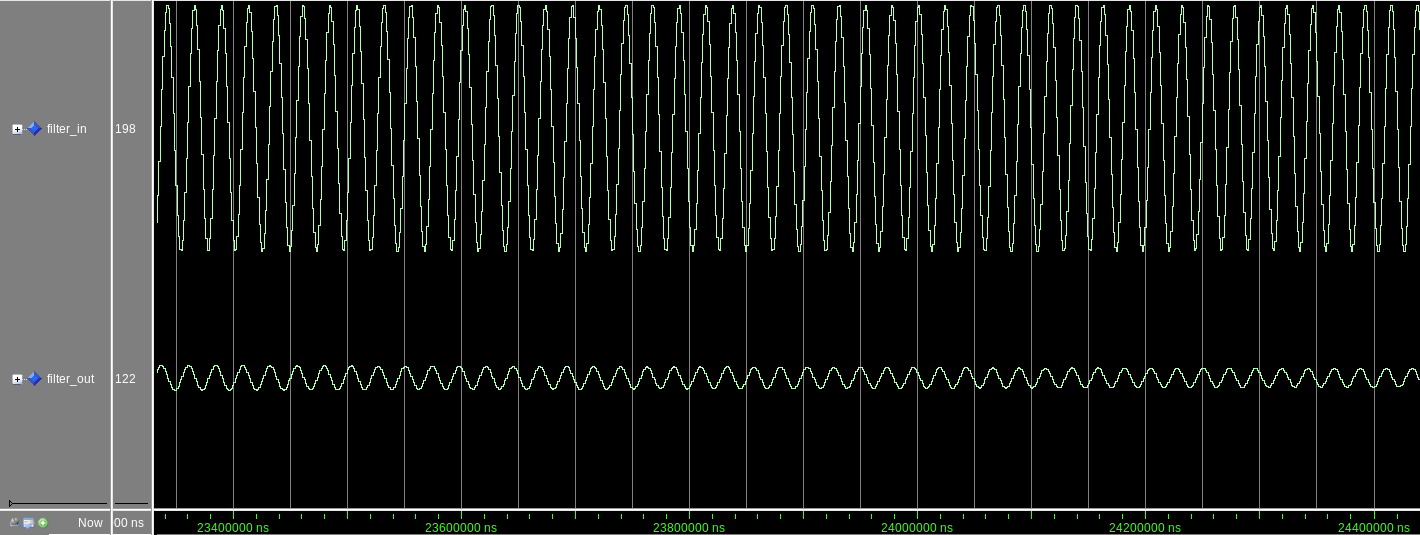
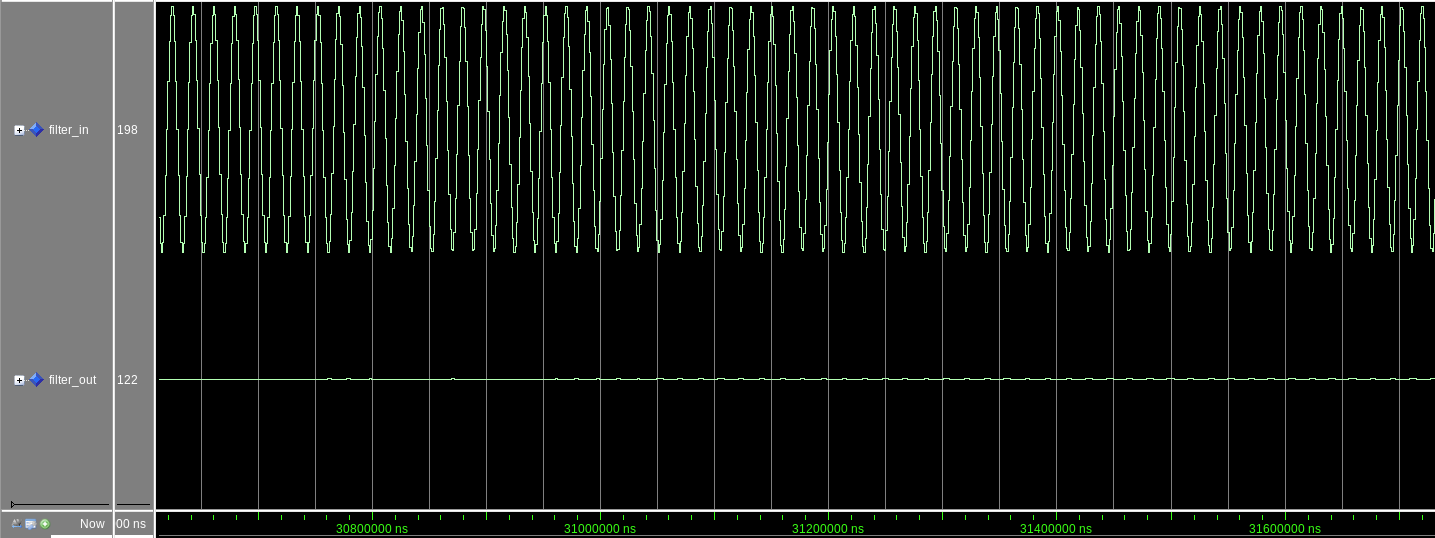
**4** -

Ici, nous allons mettre les captures d’écran montrant le bon comportement du filtre lorsqu’on lui applique une sinusoïde de fréquence variable en entrée :



Voici quelques captures sur des zones ciblés, tout d’abord en BF, puis en moyenne fréquence, et enfin en HF et très haute fréquence :





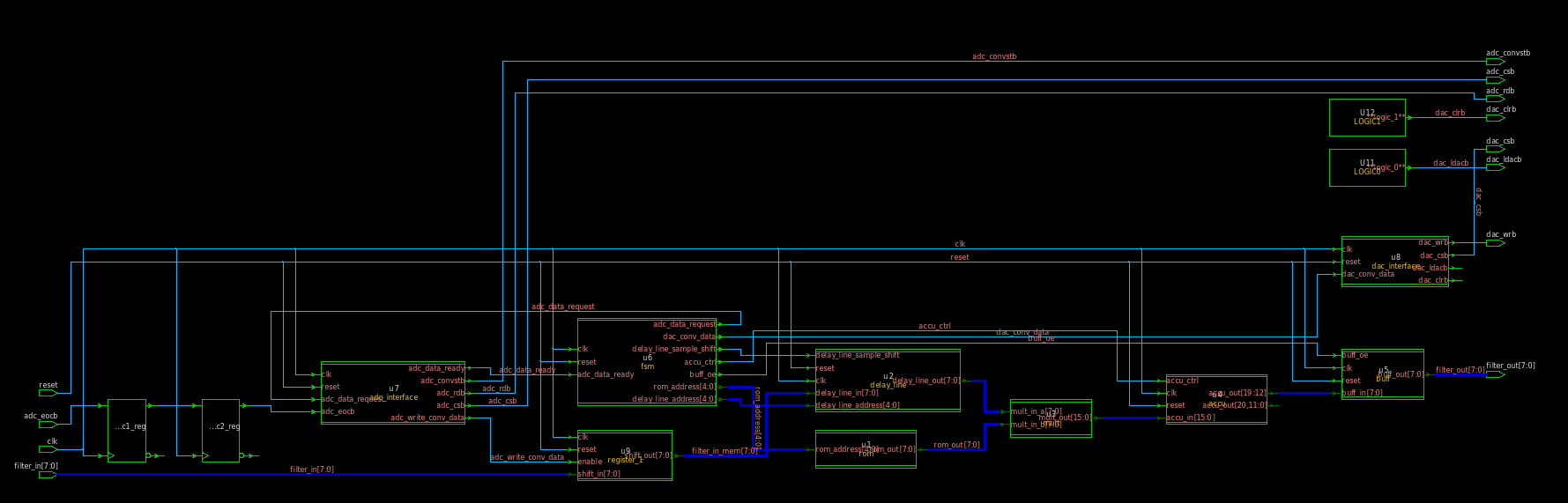
On remarque que le filtre se comporte exactement comme attendu (c'est-à-dire comme un filtre passe bas). Il laisse passer les basses fréquences et diminue l’amplitude du signal en sortie au fur et à mesure que la fréquence du signal d’entrée augmente.

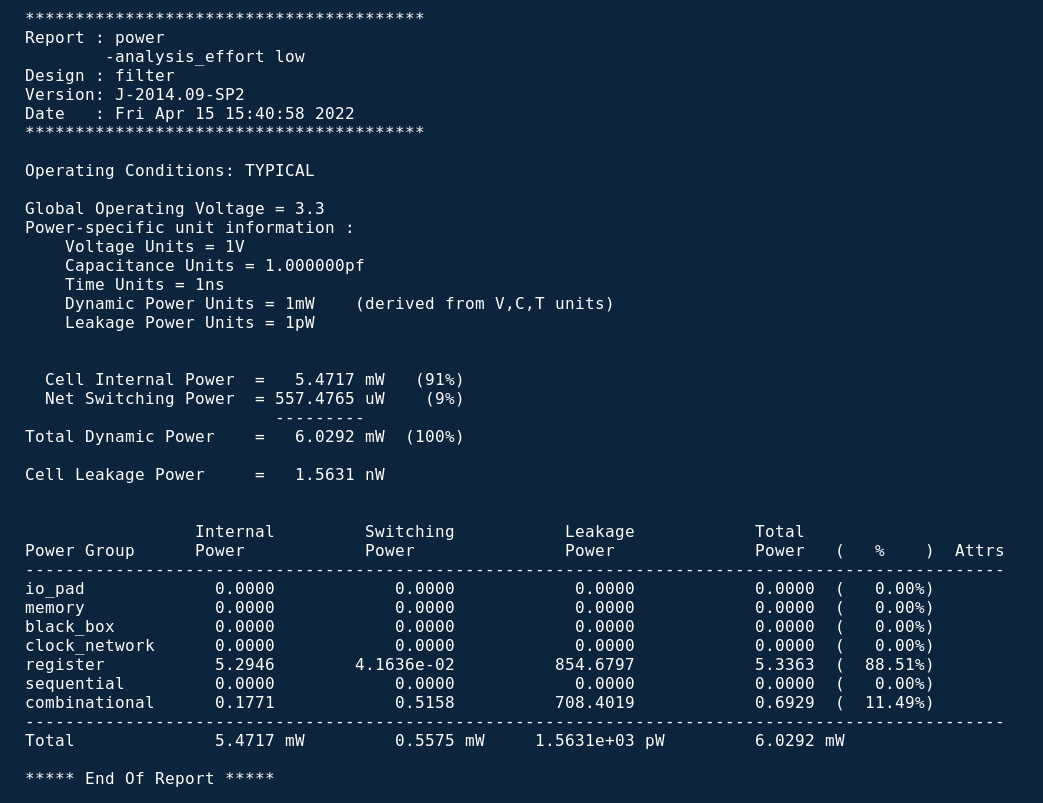
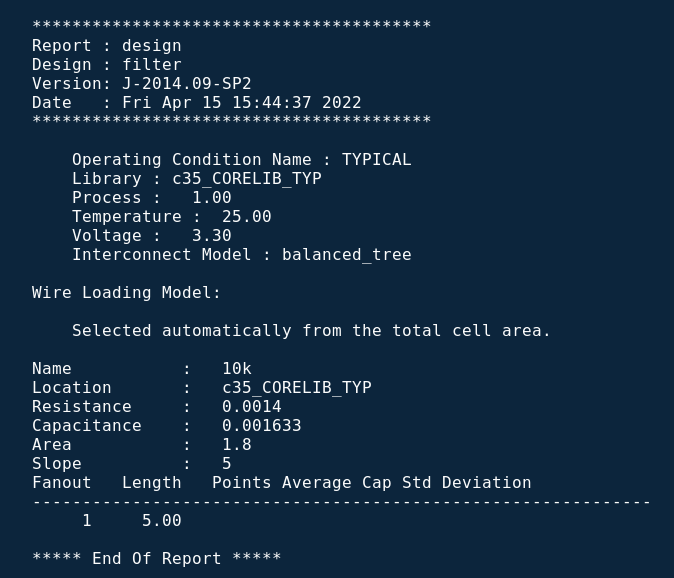
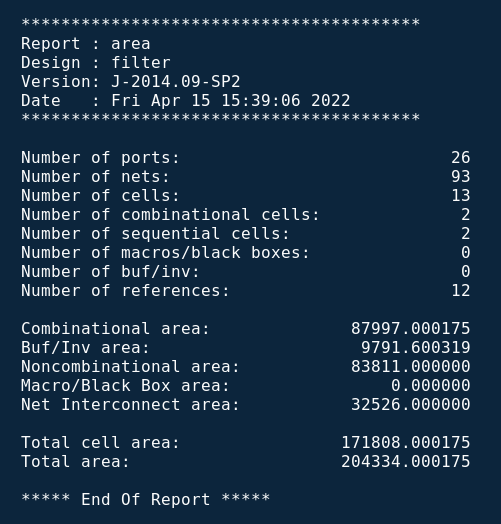
**5** -

A l’aide du logiciel Spyglass, nous avons fait une série de tests permettant de valider que notre modèle est correctement synthétisable. Ces tests nous ont notamment permis d’identifier et de corriger quelques erreurs dans la taille de certains ports.

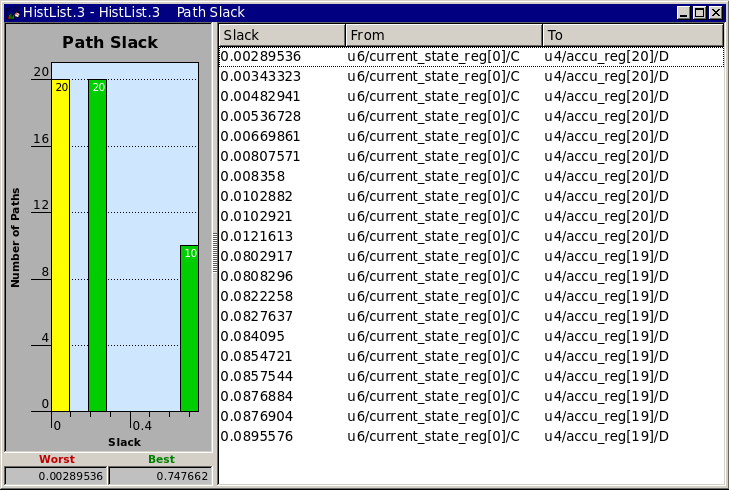
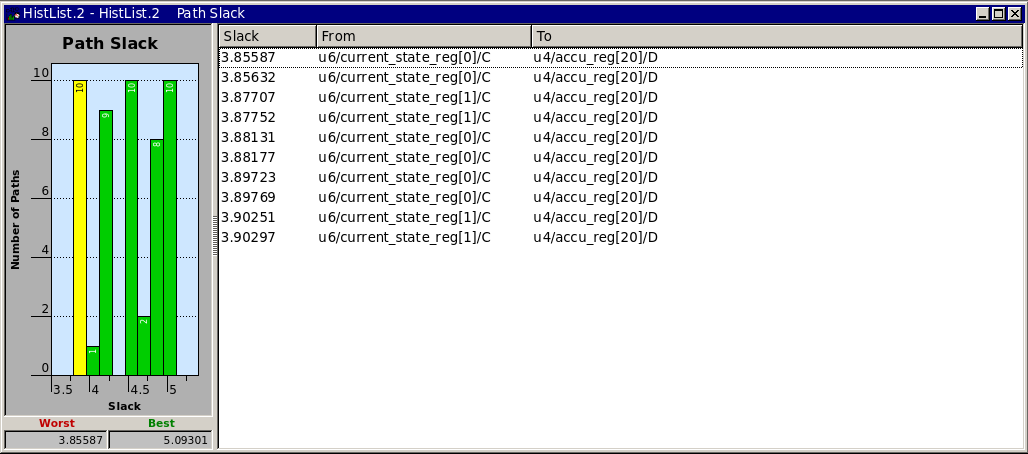
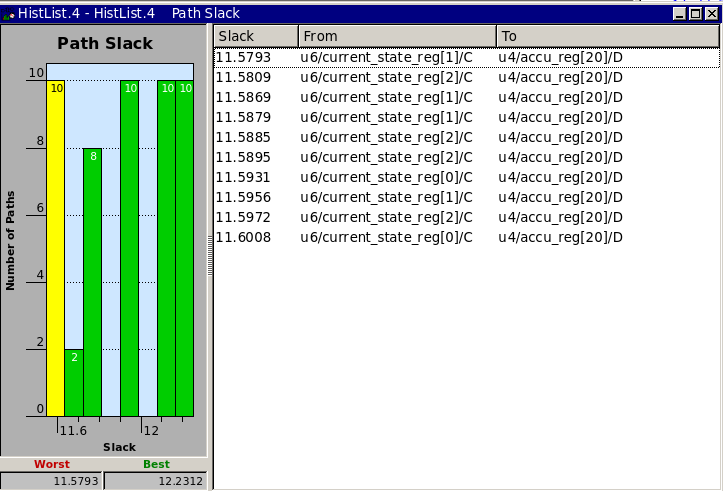
**6** -

Nous avons décidé de réaliser une implémentation ASIC de notre filtre. Nous avons donc dans un premier temps réaliser une synthèse grâce à l’outil DesignVision. Voici le schéma que nous avons alors obtenu :

Nous avons fixé en entrée du filtre un signal d’horloge à une fréquence de 50 MHz (soit une période de 20 ns), et voici ce que nous avons obtenu pour des conditions *typical* :



Nous avons obtenu les chemins suivants dans les cas best, typical et worst :



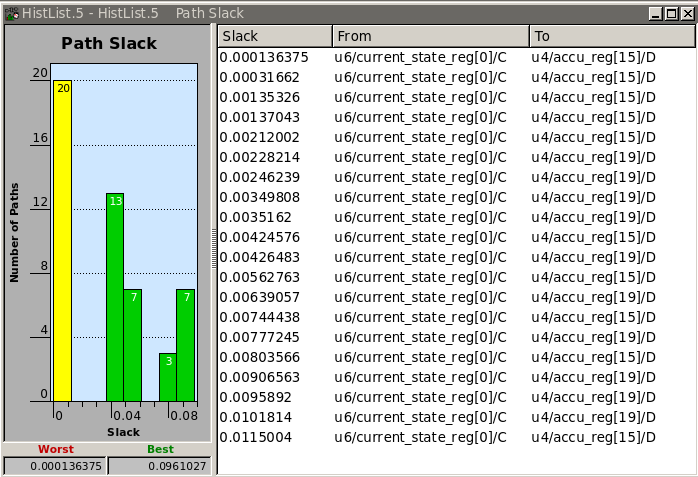
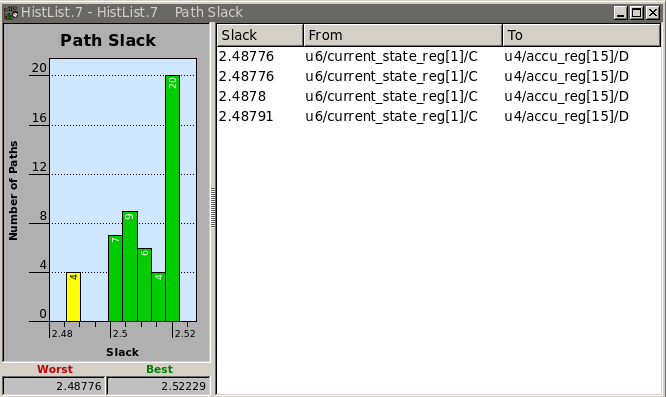
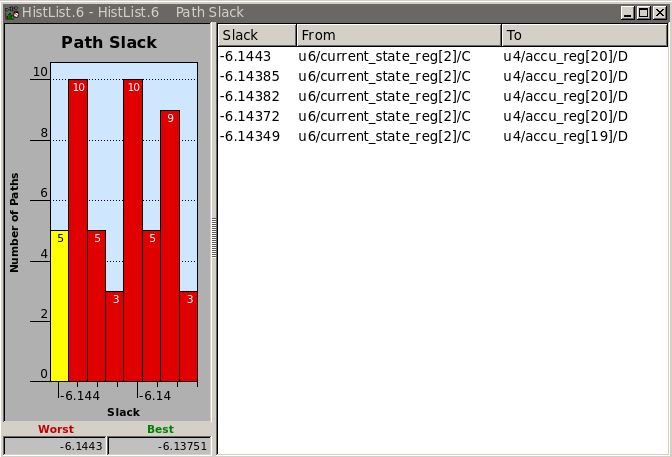
On remarque que même dans les pires conditions d'exécution, le pire des chemins rentre dans la période de 20 ns.

Nous n’avons malheureusement pas pu aller au bout de notre implémentation en raison d’un problème avec le logiciel Innovus, et nous n’avons donc pas pu réaliser la partie placement & routage, qui semblait pourtant très intéressante. Nous nous sommes donc concentrés sur l’optimisation de notre implémentation du filtre.

**Partie optimisation du filtre**

**1.a** -

Nous avons cette fois injecter à notre filtre une horloge de fréquence 100 MHz (10 ns de période), et nous avons obtenue les chemins suivants dans les cas best, typical et worst (dans l’ordre sur les images) :



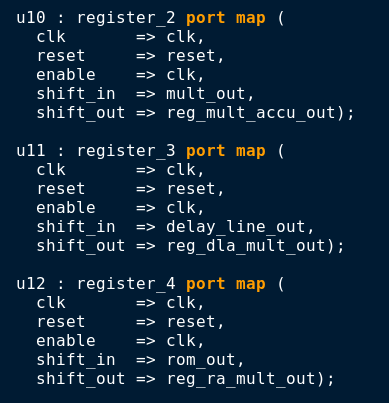
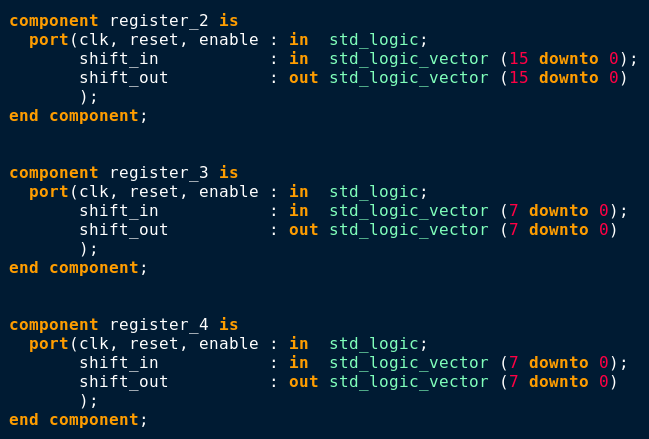
On peut voir qu’avec une période de 10ns, les cas d'exécution best et typical fonctionnent, mais que les cas worst ne fonctionne pas, c’est à dire qu’ils nécessitent une période de plus de 10ns.

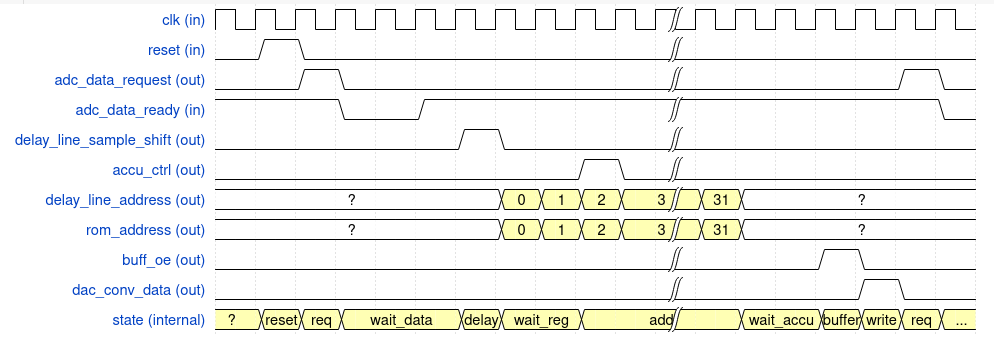
**1.b** -

En observant les pires chemins (CP ou Critical Path), on remarque que c’est notamment le *multiplieur* qui demande beaucoup de ressources et qui prend le plus de temps. Nous n’avons malheureusement pas enregistré l’image montrant le pire chemin, mais nous avons pu voir que ce dernier demandait plus de 10 ns à traverser rien que pour lui.

**2** - Pour compenser, nous avons utilisé l’optimisation par pipeline, qui consiste à paralléliser les différentes opérations pour les rendre indépendantes. Pour ce faire, nous avons ajouté 3 registres dans le but d’isoler *Mult* : un entre *RAM\_Coeff* et *mult* (*register4*), puis un entre *Delay\_line* et *Mult* (*register3*) et un entre *Mult* et *accu* (*register2*).Ces trois registre permettent de découpler le multiplicateur des autres composants. Cette technique rajoute des cycles à l'exécution, mais comme chaque cycle est beaucoup plus court, cela permet un gain de temps global.

Pour les implémenter, nous nous sommes inspirés du registre entre adc\_interface et le filtre :



Nous avons également dû modifier la fsm afin qu’elle respecte le chronogramme suivant (que nous avons écrit) :

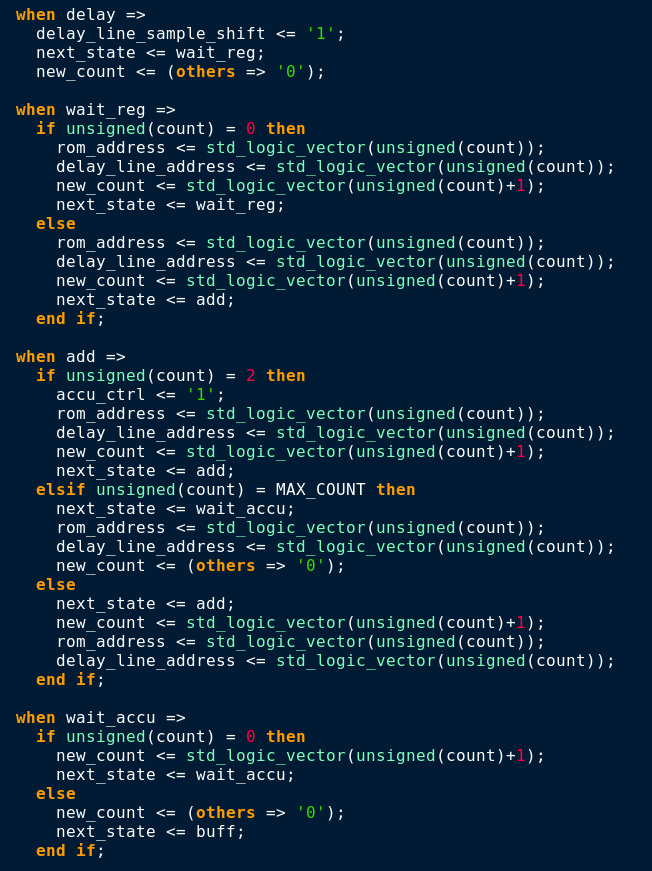
Nous avons ajouté les états *wait\_reg* et *wait\_accu*.

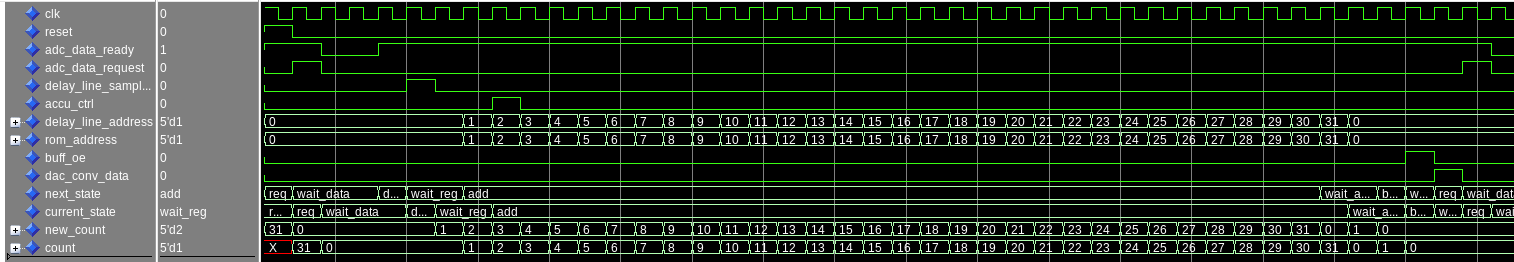
*wait\_reg* correspond au moment où *register3* et *register4* sont remplis mais pas *register2*. (premier échantillon)

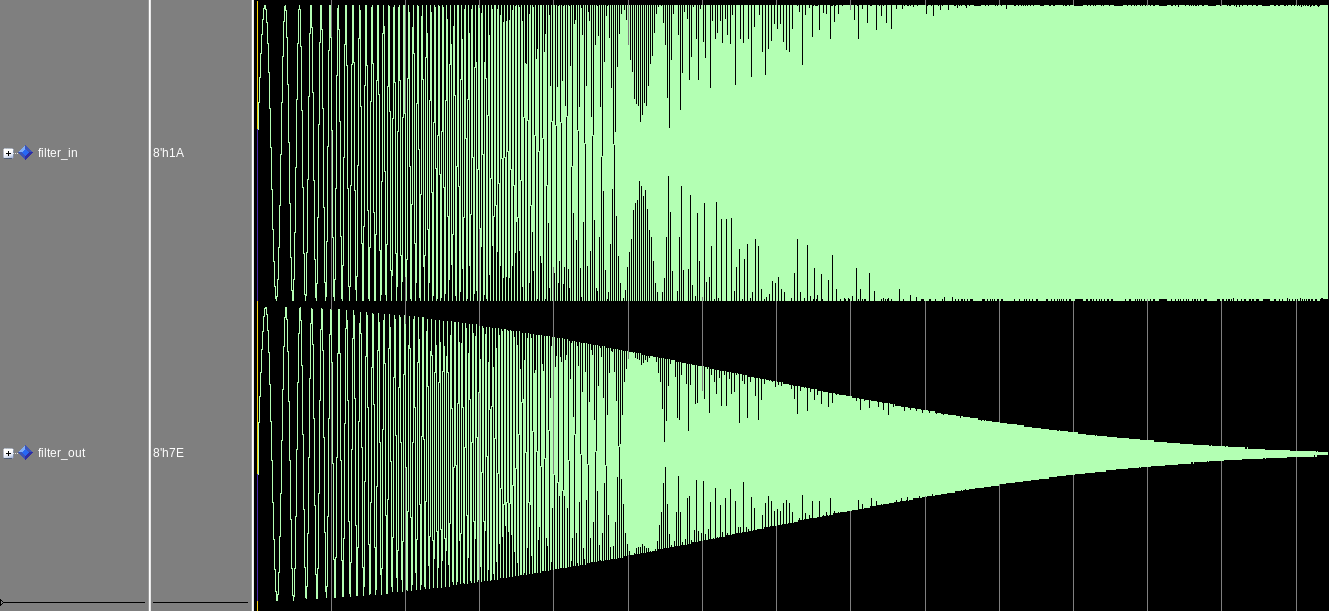
*wait\_accu* correspond au moment ou *register2* est rempli mais pas *register3* et *register4* (dernier échantillon).

Ainsi, deux cycles sont ajoutés : un lorsque *register2 et register3* sont vides et un lorsque *register4* est vide.

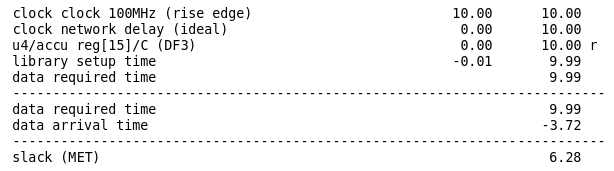
Voici les changements que nous avons effectués dans le code pour ceci :



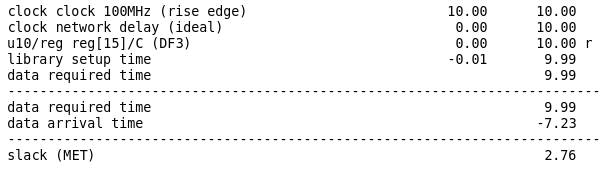
Nous avons obtenu le chronogramme suivant et nous avons vérifié que le filtre marche toujours : 

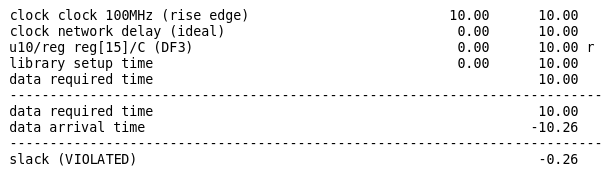


On constate bien que le filtre est toujours opérationnel après ces modifications. Après ceci, nous avons regardé les durées des chemins pour les conditions best :



typical :

et critical :



On voit que malgré la présence du pipeline, on ne peut pas passer en dessous des 10 ns dans les pires conditions d'exécutions, mais nous n’en sommes plus très loin (il ne manque plus que 0.26 ns). Cela est dû au fait que le composant Mult n’est pas assez rapide. Pour résoudre ce problème, il faudrait donc changer complètement le composant multiplicateur.