Implantation d'un filtre numérique au sein d'un FPGA

Lors de ce premier exemple nous allons implanter un filtre numérique non récursif à réponse impulsionnelle finie (FIR) ; ce filtre relativement simple sera de type passe bas et ne comprendra que six coefficients ; cette première approche nous permettra de voir un certain nombre de problèmes liés à l'implantation des filtres au sein d'un FPGA. Nous utiliserons pour nos expérimentations le FLEX10K20 (ou 10K100 suivant les versions) de la carte de développement DLP de chez Altera. Cette carte sera associée à une carte auxiliaire comprenant les CNA, CAN et filtre anti-repliement.
On trouvera en annexe 1 des rappels de base sur les filtres numériques, en annexe 2 des éléments d'utilisation du logiciel ScopeFIR qui nous permettra de calculer les coefficients du filtre, en annexe 3 des rappels sur le codage et les opérations sur les nombres binaires.

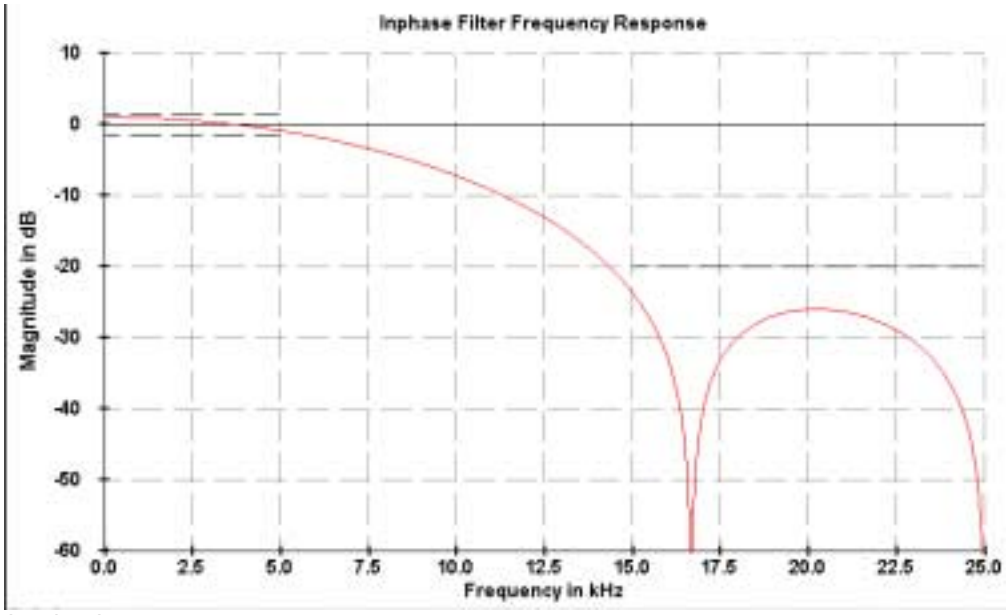
1 Implantation d'un filtre passe bas

1.1 Calcul des coefficients du filtre

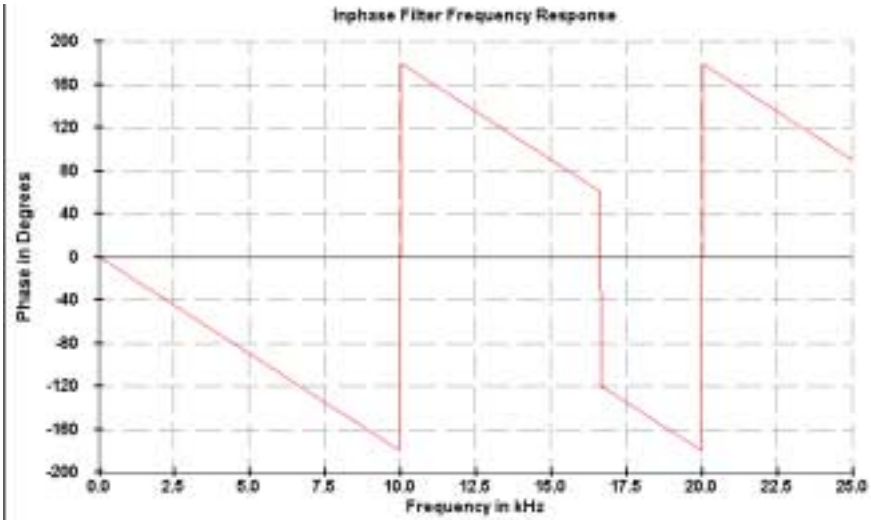
A l'aide du logiciel ScopeFIR (voir annexe 2), on détermine les coefficients du filtre passe bas pour une fréquence d'échantillonnage de 50 kHz, une bande passante à –3 dB de 5 kHz et une bande atténuée de 20 dB de 15 kHz, 6 coefficients de 9 bits (ce nombre sera justifié ultérieurement), la grille étant fixée à 16.
On arrive alors à :

h0=h5= $0{,}01171875_D$	$=00\ 0000\ 0110_B$	$=006_H$
h1=h4= $0{,}1875_D$	$=00\ 0110\ 0000_B$	$=060_H$
h2=h3= $0{,}35546875_D$	$=00\ 1011\ 0110_B$	$=0C6_H$

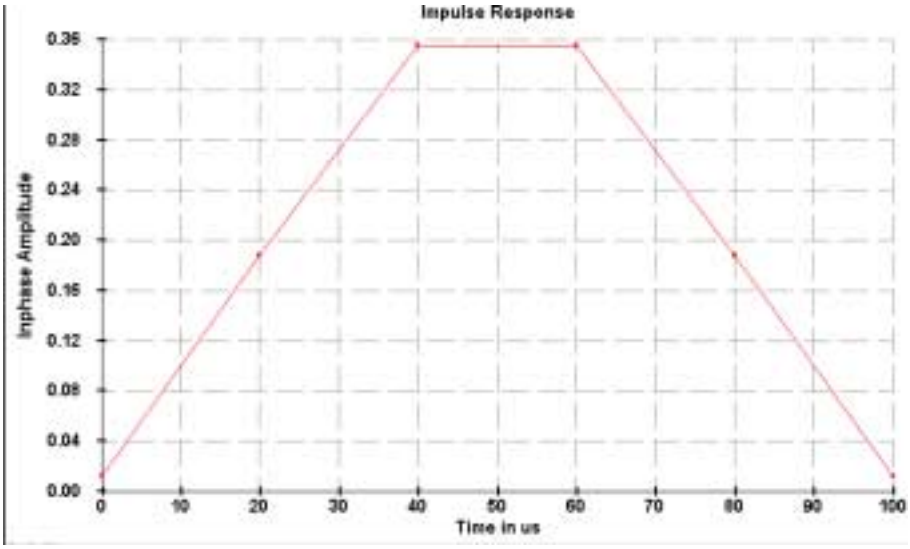
La courbe de gain donnée par ScopeFIR est alors la suivante :



et la courbe de phase :

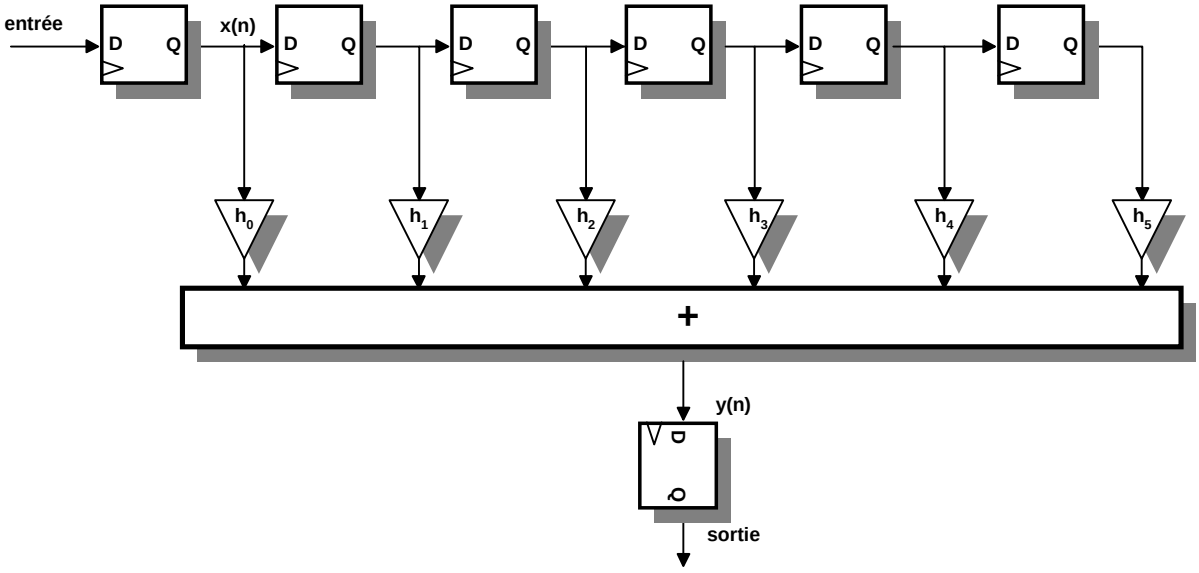


la réponse pulsionnelle correspondant à la valeur des échantillons :



1.2 Schéma d'implantation du filtre

Le schéma de base d'implantation d'un tel filtre est le suivant :



Un premier registre 8 bits (les convertisseurs de la carte auxiliaire sont de 8 bits) fait l'acquisition de la sortie du CAN et les registres suivants mémorisent l'information. Les multiplieurs et l'additionneur créent alors la fonction recherchée, à savoir :

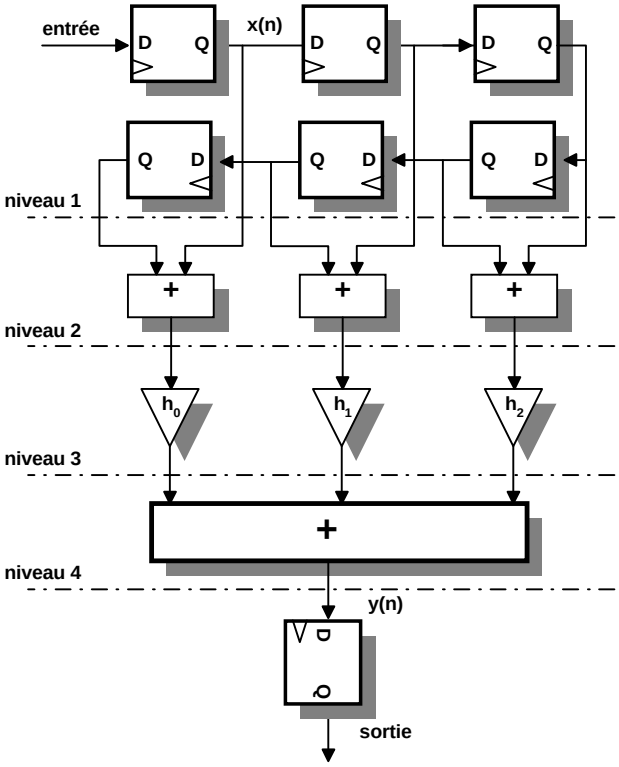
$$y(n)=h0.x(n)+h1.x(n-1)+h2.x(n-2)+h3.x(n-3)+h4.x(n-4)+h5.x(n-5)$$

Le dernier registre permet d'éviter d'envoyer sur le CNA une information erronée, due au temps de propagation aléatoire de la couche combinatoire composée des multiplieurs et additionneurs.

Cependant la symétrie des coefficients permet d'écrire l'équation de la manière suivante :

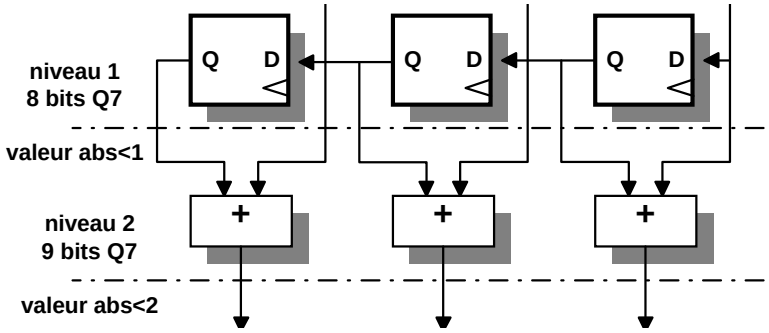
$$y(n)=h0\,[.x(n)+x(n-5)] + h1\,[.x(n-1)+x(n-4)] + h2\,[.x(n-2)+x(n-3)]$$

Cette écriture divise par deux le nombre de multiplications, et donc le nombre de multiplieurs, éléments particulièrement gourmand en ressource et ralentissant le système. On arrive alors au schéma suivant :



1.3 Détermination du nombre de bits des bus

L'acquisition et la restitution du signal se fait sur 8 bits (nombre de bits des convertisseur), en binaire décalé. Les opérations nécessaires au filtrage étant signées, on passera du binaire décalé au complément à 2 à l'entrée et on fera l'opération inverse en sortie (voir annexe 3).
Le plus simple est de considérée que les données d'entrée sont codées en Q7 sur 8 bits, le MSB représentant le signe, les 7 bits de poids faibles représentant des puissances négatives de deux (2^{-1}, 2^{-2}, 2^{-3}…). De cette manière, la valeur absolue des signaux au niveau 1 schéma précédent seront inférieure à l'unité.



L'addition de deux nombres de 8 bits donne un nombre sur 9 bits. Au niveau 2 du schéma précédent, la sortie des additionneurs nous impose un signal dont la valeur absolue est inférieure à 2, codé sur 9 bits en Q7 (1 bit de signe, 1 bit de poids 2^0, 6 bits de poids 2^{-1}, 2^{-2}, 2^{-3}…).

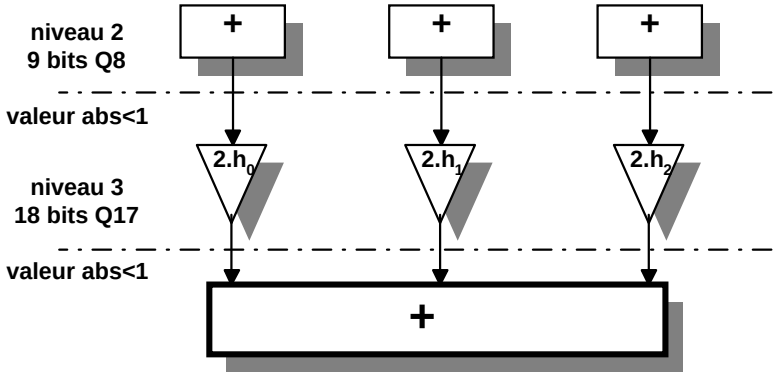
En faisant artificiellement une division par 2 de ces signaux, ce qui revient à considérer le codage comme du Q8 sur 9 bits, les sorties des multiplieurs ne dépasseront pas l'unité (en valeur absolue), les coefficients du filtre étant en effet tous inférieurs à l'unité.

On peut remarquer également que le bit de poids fort de la valeur absolue (c'est à dire le deuxième bit en partant du MSB en CP2) est nul pour tous les coefficients. On peut donc compenser la division par 2 introduite précédemment par une multiplication par deux (décalage à gauche) de tous les coefficients.

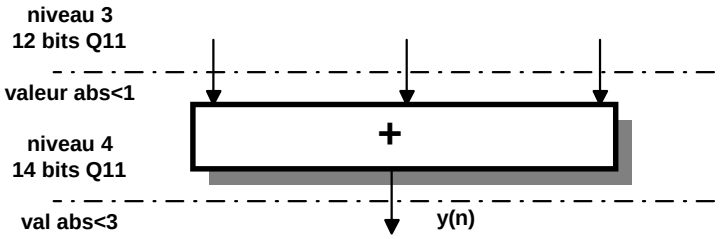
A ce niveau, on peut donc soit :

- calculer les coefficients du filtre sur 10 bits, puis effectuer la multiplication, par 2 ; le LSB du résultat sera non significatif et donc supprimé, ce qui sera sans incidence dans notre codage 9 bits en Q8. On pourra alors garder la précision optimale en sortie des additionneurs (niveau 2) sur 9bits ;
- effecteur le calcul des coefficients sur 9 bits et perdre le bit de poids faible pour les mêmes raisons que précédemment, les coefficients étant alors codés sur 8 bits en Q7. On ne gardera alors de la sortie des additionneurs au niveau 2 que les 8 bits de poids fort, perdant ainsi un peu en précision, mais allégeant la structure de l'ensemble.

Nous opterons pour la première solution.

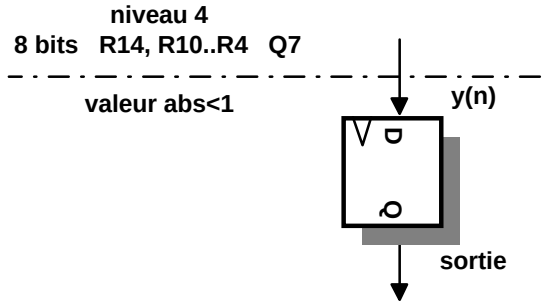


Au niveau 3, les signaux sont en valeur absolue inférieurs à l'unité, puisque résultant de la multiplication de deux nombres eux-mêmes de valeur absolue inférieur à l'unité. D'autre part la multiplication de deux nombres sur B bits donne un résultat sur 2B bits. Un résultat exprimé sur moins de bits donne une valeur erronée ou perd en précision suivant le codage.
Dans notre cas le codage est sur 18 bits en Q17 (1 bit de signe, 17 bits pour les puissances de deux inférieures à l'unité). La sortie se faisant sur 8 bits au niveau du CNA, il n'est pas nécessaire de garder cette précision, aussi tronquerons-nous la sortie des multiplieurs sur 12 bits afin d'alléger l'architecture ; le codage sera alors du Q11 (1 bit de signe, 11 bits pour les puissances négatives de 2).



La sortie du dernier additionneur qui fait la somme de trois signaux de valeur absolue inférieures à l'unité (codage Q11 sur 12 bits) donnera un résultat inférieur à 3 en valeur absolue, donc théoriquement un nombre sur 14 bits codé en Q11 (1 bit de signe, 2^{1}, 2^{0}, 2^{-1},2^{-2}…2^{-11}.).

On peut remarquer cependant que notre filtre a un gain unitaire dans la bande passante, en conséquence de quoi, si le signal d'entrée est inférieur à 1 (codage sur 8 bits en Q7 de l'entrée), la sortie ne peut dépasser l'unité. On note en effet que la somme des coefficients du filtre est inférieure à l'unité (cette somme serait la sortie dans le cas d'un signal continu –presque- unitaire). Les bits de poids 21 et 20 en sortie de l'additionneur au niveau 4 seront donc théoriquement toujours à 0,



On ne gardera donc de 14 bits R13 à R0 en sortie de l'additionneur au niveau 4 pour envoyer vers la sortie, le bit de signe R13 et les bits R10 à R4.

Remarque : une erreur de calcul du à l'arrondi des résultats peut cependant introduire dans ce cas une sortie erronée.

1.4 Implantation des éléments

Nous avons maintenant besoin pour réaliser notre filtre d'implanter trois types de composants dans le FPGA : des registres 8 bits, des multiplieurs et des additionneurs. Les deux premiers existent sous forme de macro-fonctions dans la bibliothèque **maxplus2\max2lib\mega_lpm** de maxplus. Il nous faudra également un convertisseur permettant de passer du binaire décalé (CNA et CAN) vers le complément à 2 (composants arithmétiques).

1.4.1 Les registres

Ils ne posent pas de problèmes particuliers. On peut implanter les registres en sélectionnant le composant **lpm_dff** de cette bibliothèque, ou en sélectionnant **lpm_ff** des composants de stockage par l'intermédiaire du « **MegaWizard Plug-in Manager…** ». On sélectionnera à chaque fois la version 8 bits avec aucun autre port que l'horloge, le bus de données en entrée et le bus de données en sorties.

1.4.2 Le convertisseur BD – CP2

Proposer et tester une structure VHDL combinatoire permettant de réaliser cette fonction dans les deux sens ; on placera le nombre de bits des entrées et sortie comme un paramètre générique, initialisé à 8. On nommera cette fonction BCD_CP2.

1.4.3 Les additionneurs

L'additionneur de la bibliothèque de maxplus ne gère pas l'addition signée, le plus simple est alors de créer nos propres fonctions en VHDL.

Créer un premier composant que l'on appellera ADD_SGN qui aura deux entrée sur B bits (B étant un paramètre générique initialisé à 9) et une sortie sur B+1 bits. On rappelle qu'en VHDL, suivant les paquetages utilisés, les grandeurs sont interprétées en binaire naturel ou en complément à 2

Tester le bon fonctionnement par une simulation fonctionnelle. Attention toutes les données du simulateur sont interprétées comme du binaire naturel ; il faudra donc faire la traduction en CP2. Par exemple, sur 3 bits, la valeur 7 en décimal ou 111 en binaire, doit être interprétée comme –1.
Aussi, est-il peut-être plus facile dans un premier temps d'imposer un faible nombre de bits afin d'interpréter facilement les résultats (ou alors créer un convertisseur BN CP2 qu'il faudra auparavant tester !).

Faire ensuite une compilation (avec circuit cible) pour un additionneur de 9 bits sur les ports d'entrées et déterminer la place occupée dans le circuit grâce au rapport de compilation. Estimer également les temps de propagation grâce à l'analyseur temporel (**Timing Analyseur**), option « **Delay Matrix** » (les autres options n'ayant pas de sens ici notre multiplieur étant purement combinatoire.

Créer maintenant un additionneur à trois entrées ; attention en VHDL l'extension du bit de signe n'est pas gérée avec trois entrées. Le nommer ADD_3_SGN, tester et évaluer les performances.

1.4.4 Les multiplieurs

L'implantation des multiplieurs est plus simple en passant par le « **MegaWizard Plug-in Manager**… » et en sélectionnant le composant lpm_mult du groupe « arithmetic ». On choisira les options suivantes :

- nombre de bits (dans notre cas 9 pour chaque entrée) ;
- laisser le logiciel ajuster lui-même le nombre de bits en sortie (18 dans notre cas) ;
- pas d'entrée de somme ;
- une des entrées est constante et on donnera sa valeur en décimal en interprétant la valeur en CP2 comme si c'était du binaire naturel ;
- multiplication signée (les données sont interprétée en CP2) ;
- implanter le multiplieur dans les EAB (Embedded Array Blocks) ;
- pas de structure en pipeline ;
- options de compilation par défaut (pas d'optimisation particulière sur la densité ou la vitesse).

Afin de se familiariser avec ce composant, faire un essais à part avec 3 bits sur les ports d'entrées dans un premier temps ; effectuer une simulation fonctionnelle avec une des entrées s'incrémentant régulièrement (l'autre étant constante) et vérifier la validité des résultats.

Faire ensuite une compilation (avec circuit cible) pour un multiplieur de 9 bits sur les ports d'entrées et déterminer la place occupée dans le circuit et les temps de propagation.

Conclure sur la possibilité d'implanter notre filtre dans le circuit et de le faire fonctionner à une fréquence d'horloge de 50 kHz.

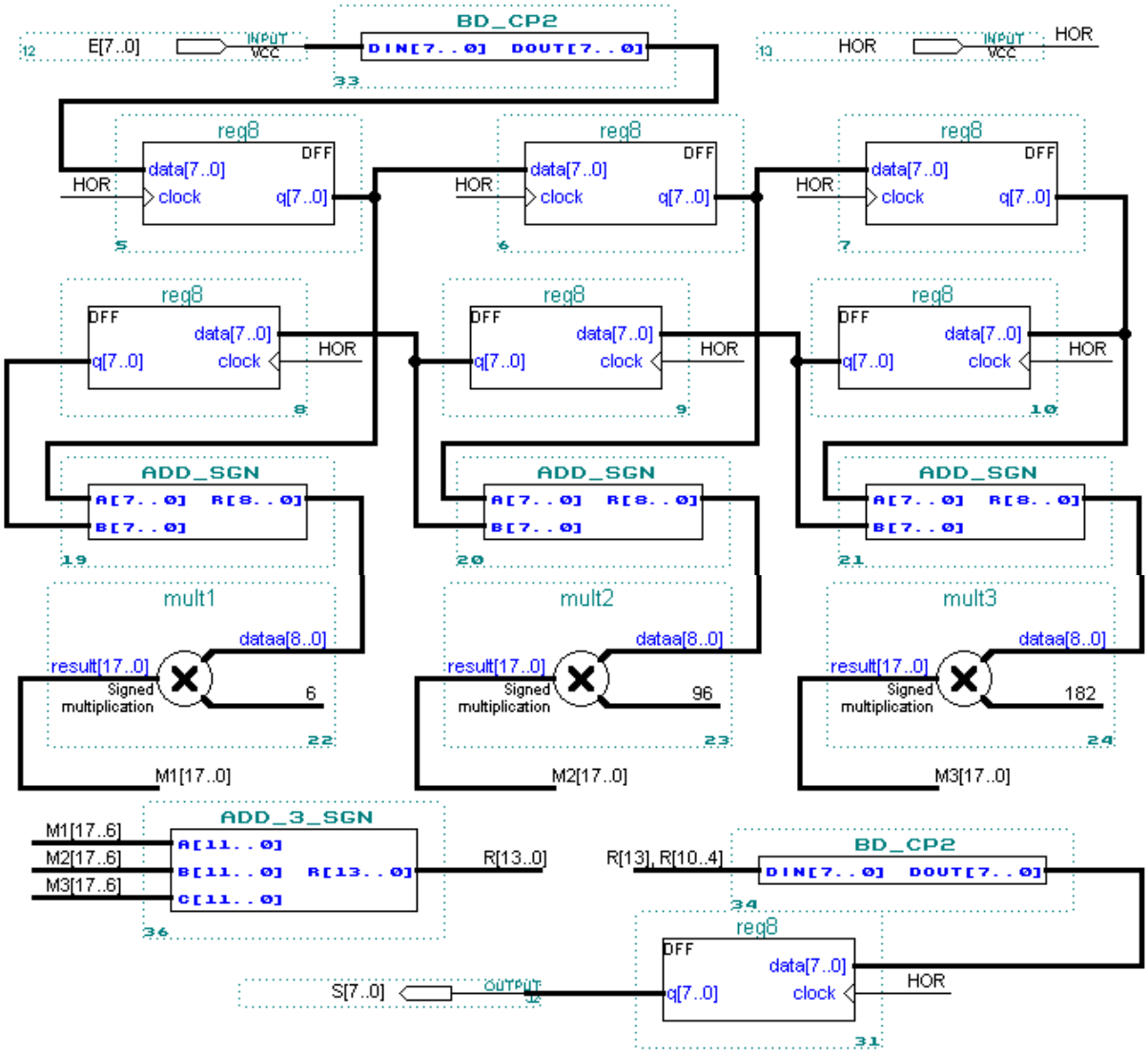
Les coefficients du filtre sont à fournir au multiplieur en décimal, en interprétant la valeur binaire du coefficient comme si elle était exprimée en complément à 2 et Q0, ce qui donne dans notre cas :

h0=h5= $0{,}01171875_D$	$=00\ 0000\ 0110_B$	$=006_H$	=> entrer 6 dans le multiplieur
h1=h4= $0{,}1875_D$	$=00\ 0110\ 0000_B$	$=060_H$	=> entrer 96 dans le multiplieur
h2=h3= $0{,}35546875_D$	$=00\ 1011\ 0110_B$	$=0C6_H$	=> entrer 182 dans le multiplieur

Remarque : nous avons trois multiplieurs à implanter, chacun multipliant par une constante différente ; il s'agira donc de trois variations créées avec le « **MegaWizard Plug-In Manager…** » de la méga-fonction lpm_mult, avec trois noms différents. Pour changer la valeur d'une constante sur un multiplieur implanter, on double clic sur le composant, on effectue le changement, puis lors du retour dans l'éditeur graphique on valide par le menu **Symbol / Update Symbol….**

1.5 Schéma complet

Voici le schéma obtenu pour notre filtre :



Faire une compilation fonctionnelle dans un premier temps, puis une compilation tenant compte du circuit cible, et vérifier la place occupée dans le circuit (rapport de compilation), ainsi que la fréquence maximale admissible pour l'horloge (« **Timing analyser** » puis « **Registered performances** »).

Remarque : ce schéma n'est pas optimal vis à vis des performances potentielles du FPGA ; on pourra consulter pour plus d'information la note d'application d'Altera AN-073 « Implementing FIR Filters in Flex Devices ».

1.6 Simulations et essais

Pour tester notre filtre, nous utiliserons la méthode vue précédemment avec la synthèse numérique directe :

- création d'un stimuli « analogique » avec Scilab et générer le fichier .vec pour le simulateur de maxplus ;
- lancer la simulation sous maxplus ;
- créer un fichier résultat .tlb, le modifier pour permettre la lecture par Scilab et afficher les résultat dans Scilab.

Le programme suivant génère un signal dont la fréquence passe de 5 kHz, à 15 kHz puis à 20 kHz, le tout pendant une durée de 3,14 µs. La possibilité est laissée de vérifier le stimuli en l'affichant dans Scilab.

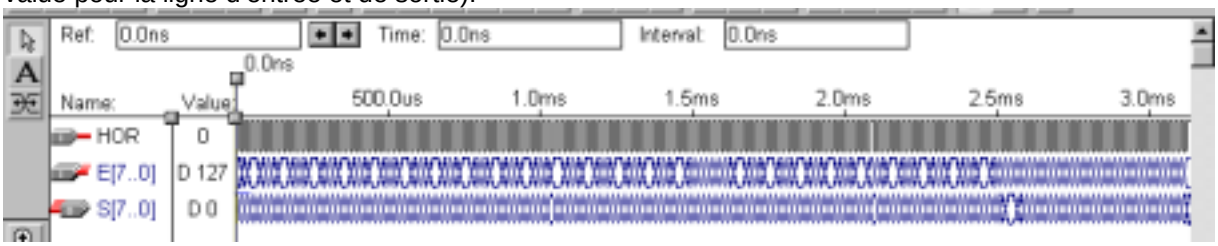
```
// initialisation et passage dans le répertoire de travail de maxplus
clear; chdir('f:\max\Stagen2\ESSAIS_FIR') ;
//
// définition des paramètres fréquence du signal modulant
// période d'horloge (en ns), durée de simulation (en ns)
F5=5e3;F15=15e3; F20=20e3; Tck=20e3; Tend=3140e3;
//
// génération du temps (en µs)
t=Tck*(0:Tend/Tck-1);
// mise sous forme de matrice colonne avec un pas de 10 µs (voir structure fichier .vec)
t2=Tck/2*(0:Tend/(Tck/2)-1)';
//
// génération du signal d'entrée
// la moité du temps à 5kHz, puis 30% à 16kHz, puis 20% à 20kHz
// amplitude crête de 127
X5=127*(1+sin(2*%pi*F5*t(1:length(t)/2)*1e-9));
X16=127*(1+sin(2*%pi*F15*t(length(t)/2+1:.8*length(t))*1e-9));
X20=127*(1+sin(2*%pi*F20*t(.8*length(t):length(t))*1e-9));
X=[X5 X16  X20 ];
//
// affichage
xbasc();xset("font size",4);
plot2d(t,X);
xtitle("signal","temps (µs)","amplitude") ;

La suite permet la création d'un fichier filtr1_scilab.vec

// création d'un fichier .vec
// à chaque période d'horloge correspond une suite 0 1 dans le fichier « .vec »
h=[0;1]*ones(1,length(t));
hor=matrix(h,2*length(t),1);
//
// dédoublement des valeurs de X, afin de suivre les 0 1 de l'horloge
XA=[1;1]*X;
XB=matrix(XA,2*length(t),1);
//
// création du fichier
fd=mopen("filtr1_scilab.vec","w");
mfprintf(fd,"\t%i> \t%i  \t%i =\t XX \n",t2, hor, XB);
mclose(fd);
```

On modifiera l'en-tête et la fin de ce fichier pour le rendre compatible avec ceux de maxplus ; pour cela on s'inspire d'un fichier .tbl créé à la suite d'une simulation « bidon » de notre filtre (dans le simulateur ou l'éditeur de chronogrammes : **File / Create a Table File**).
Le fichier .vec finale a l'allure suivante

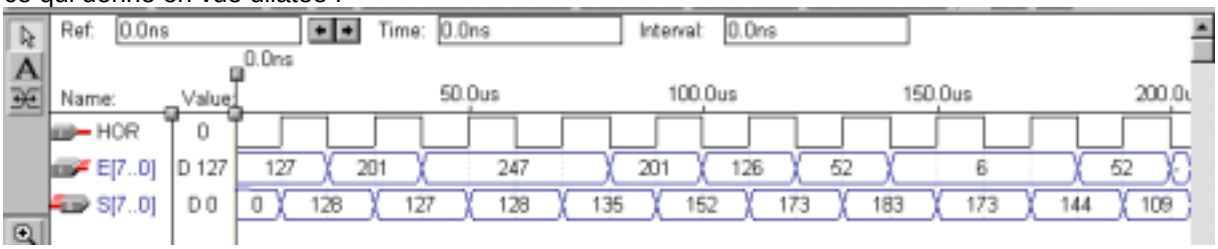
```
GROUP CREATE E[7..0] = E7 E6 E5 E4 E3 E2 E1 E0 ;
GROUP CREATE S[7..0] = S7 S6 S5 S4 S3 S2 S1 S0 ;
INPUTS HOR E[7..0]\DEC ;
OUTPUTS S[7..0]\DEC ;
UNIT ns ;
RADIX HEX ;
PATTERN

0> 0  127 = XX
10000> 1  127 = XX
20000> 0  201 = XX
30000> 1  201 = XX
40000> 0  247 = XX
50000> 1  247 = XX
```

```
.....
.....
3130000> 1  127 = XX
3140000.0> X XXX = XXX
;
```

Après déclarer le nouveau fichier .vec au simulateur de maxplus (dans l'éditeur de chronogrammes, le fichier .scf étant actif : **File / Import Vector File**), lancer la simulation et vérifier que les résultats affichés sont bien en décimal (sinon imposer la notation décimale par un double clic dans la colonne value pour la ligne d'entrée et de sortie).



ce qui donne en vue dilatée :



Créer un fichier résultat .tbl (dans le simulateur ou l'éditeur de chronogrammes : **File / Create a Table File**).

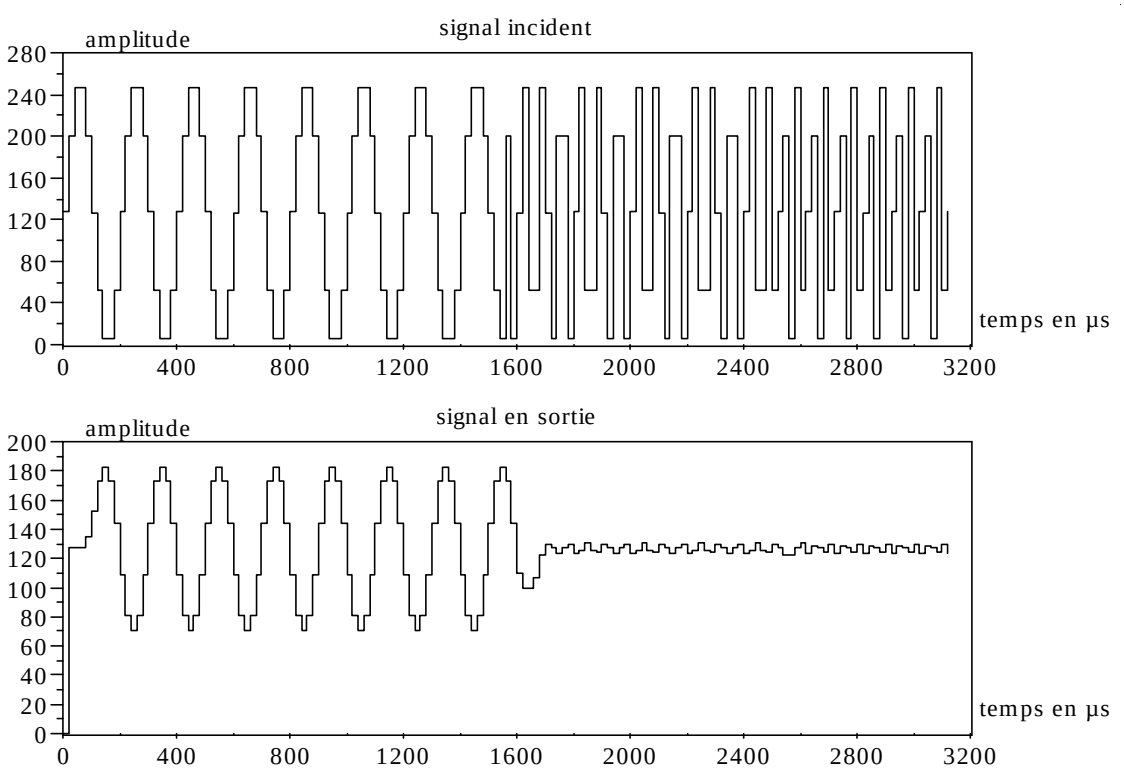
Modifier le fichier en supprimant l'en-tête et la fin, modifier éventuellement son nom (fir1-result_mod.vec dans l'exemple).

```
    0.0> 0 127 = 000
 10000.0> 1 127 = 000
 10000.1> 1 127 = 128
 20000.0> 0 201 = 128
 30000.0> 1 201 = 128
……
……
3120000.0> 0 127 = 123
3130000.0> 1 127 = 123
3130000.1> 1 127 = 129
```

Lire et afficher les résultats sous Scilab avec le programme suivant :

```
// lecture d'un fichier « .tbl » issus de Max+plus
clear;
//
// changement du répertoire courant vers celui de max+plus
// ici f:\max\stagen2\essais_fir
chdir('f:\max\stagen2\essais_fir') ;

// ouverture du fichier fir1-result_mod.tbl
fd=mopen("fir1-result_mod.tbl","r");
//
[n,temps,hor,X,Y ]=mfscanf(-1,fd,"%f> %f %f =  %f");
mclose(fd);
//
```

```
//
IN=X(1:3:length(X))';
OUT=Y(1:3:length(Y))';
//
t=20*(0: length(IN)-1);
//
xbasc(); xset("font size",4);
xsetech([0,0,1,1/2]); plot2d2(t,IN); xtitle("signal incident","temps en µs","amplitude");
xsetech([0,1/2,1,1/2]); plot2d2(t,OUT); xtitle("signal en sortie","temps en µs","amplitude");
```



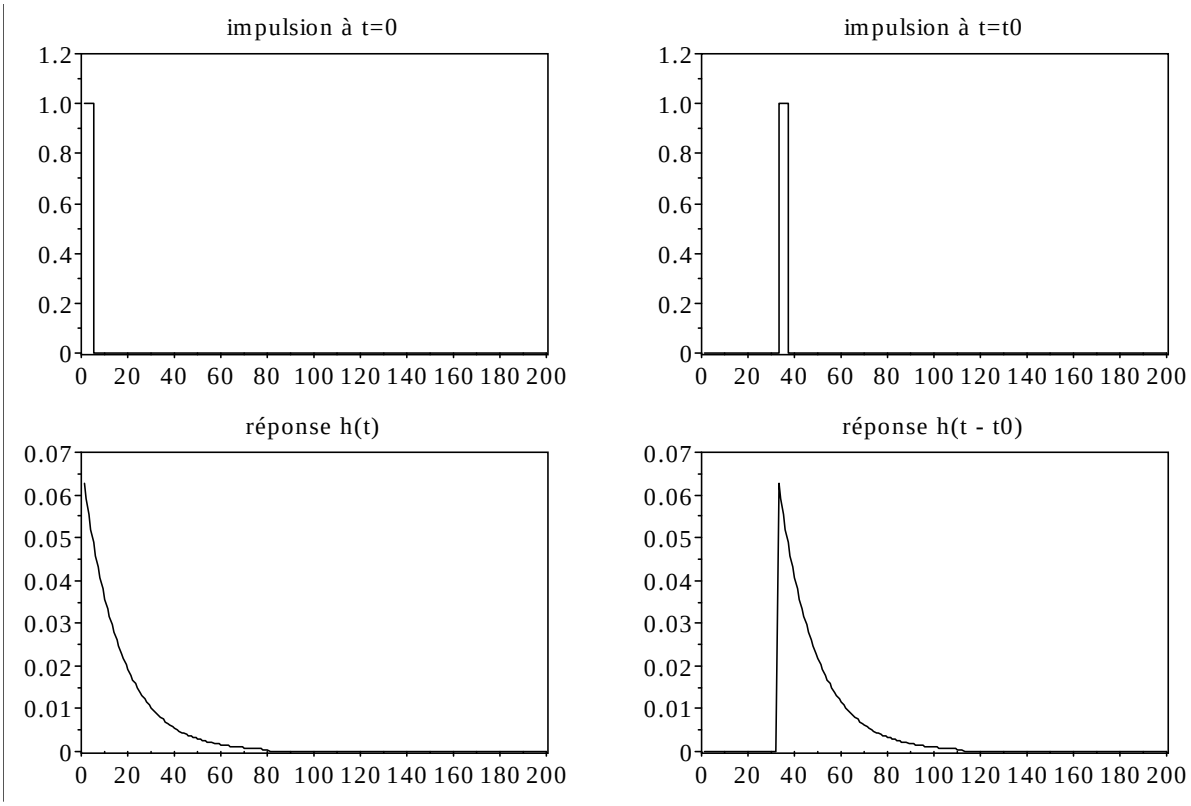
Le signal d'entrée est bien celui que nous avons imposé avec Scilab :il est centre en milieu de dynamique sur 127, son amplitude crête est de 127, sa fréquence est dans un premier temps de 5 kHz (soit 10 échantillons par périodes avec un échantillonnage de 50 kHz), puis passe à 15 kHz (soit 3,33 échantillons par périodes, d'où le battement), puis à 20 kHz (soit 2.5 échantillons par secondes, d'où de nouveau un battement).
Lorsque l'entrée est à 5 kHz, on mesure une sortie d'amplitude crête 57 environ (lu en dilatant les échelles dans l'éditeur graphique de Scilab), ce qui donne une atténuation de 2,2 soit –6.9 dB, ce qui est supérieur à l'atténuation attendue (-3 dB sur le gabarit, -1.2 sur le gain tracé par ScopeFIR). Le déphasage mesuré est également plus important (-180° environ au lieu de –90°).
Lorsque le signal est de fréquence 15 et 20 kHz, l'atténuation est cette fois de 30 dB (au lieu des 25 prévus).

Ajouter maintenant à la description le bloc de commande du CAN et de gestion de l'horloge, assigner le circuit cible et affecter les broches d'entrées sorties, programmer le circuit et tester.

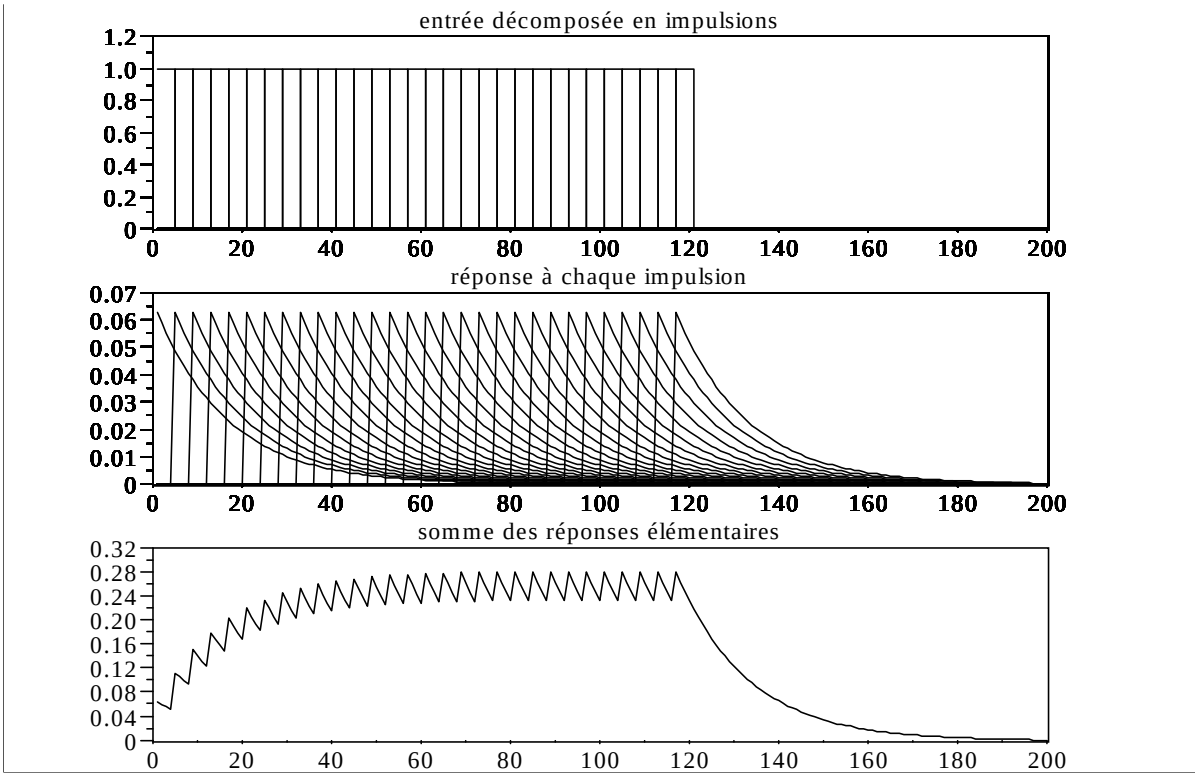
Annexe 1 : notions de base en filtrage numérique

Approche intuitive

Un quadripôle est caractérisé par la réponse qu'il produit à une impulsion. Connaissant cette réponse, il est alors possible si le quadripôle est linéaire, de déterminer la réponse à n'importe quelle sollicitation. On peut en effet remarquer q'un signal d'entrée quelconque peut être vu comme une suite d'impulsions plus ou moins grandes collées les unes aux autres. Le signal de sortie sera donc la somme des différentes réponses aux impulsions.
Les figures suivantes illustrent, de manière grossière afin de pouvoir observer l'algorithme de calcul, cette opération pour un système du premier ordre (filtre RC passe bas par exemple).
On présente dans un premier temps la réponse du système à une impulsion à t=0, puis à la même impulsion retardée de t0



En appliquant une entrée en créneau décomposée en somme d'impulsions élémentaires, on peut alors observer la réponse globale du système.



On retrouve bien la forme de réponse attendue pour un premier ordre, à deux détails près : une ondulation importante et une asymptote qui ne tend pas vers 1. Ces deux imperfections résultent de la volonté de prendre des impulsions larges afin d'observer les détalls du calcul.

Cette opération qui consiste à faire la somme toutes les réponses impulsionnnelles, décalées en fonction de la position de l'impulsion d'entrée, et pondérées par l'amplitude de celle-ci est un produit de convolution.
Dans le cas d'un système numérique, les signaux étant échantillonnés à une fréquence $F_E=1/T_E$, ils ne sont connus qu'aux instant nT_E (n étant un entier) et écrit sous forme x(n). Le produit de convolution s'écrit alors comme une somme de valeur discrète.

Supposons un quadripôle dont la réponse impulsionnelle h(nTE) soit composée de trois échantillons h0, h1, et h2, le autre étant négligeables devant le quatum du système (la réponse d'un quadripôle stable doit tendre vers 0). Pour un signal x arrivant à l'instant t=0,nous obtenons alors le signal y en sortie tel que :

y(0)= h0 x(0)
y(1)= h0 x(1) + h1 x(0)
y(2)= h0 x(2) + h1 x(1) + h2 x(0)
y(3) =h0 x(3) + h1 x(2) + h2 x(1) + h3 x(0)
etc…

ce qui peut s'écrire encore :

$$y(n) = \sum_{k=0}^{N} x(k).h_{n-k}$$

La première expression montre sans ambiguïté la commutativité du produit de convolution ; on peut alors écrire :

$$y(n) = \sum_{k=0}^{N} x(k).h_{n-k} = \sum_{k=0}^{N} h_k.x(n-k)$$

C'est cette dernière partie de l'équation qui va nous permettre d'implanter un filtre au sein d'un système numérique.

Remarque : en analogique, le produit de convolution s'écrit :

$$y(t) = \int_{-\infty}^{+\infty} x(\tau)\ \ h(t-\tau)\,d\tau = x(t) * h(t)$$

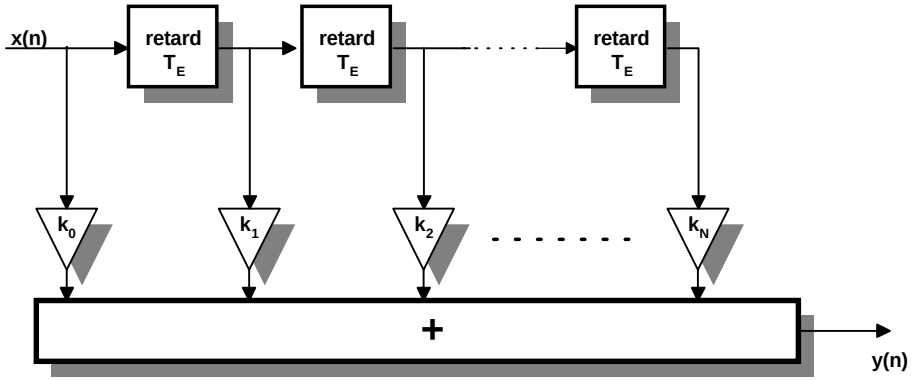
A cette expression peu adaptée au calcul analytique, on préfère utiliser en analogique, sa transformation de Fourrier qui est la fonction de transfert du quadripôle. Cette transformation changeant du même coup un produit de convolution en un produit simple, les calculs en sont grandement facilités.

Filtre récursifs et non récursifs

Filtres non récursifs

L'équation que nous venons d'établir va nous permettre d'implanter un filtre dit non récursif : la sortie ne dépend que de l'entrée et de la réponse impulsionnelle :

$$y(n) = \sum_{k=0}^{N} h_k . x(n-k)$$

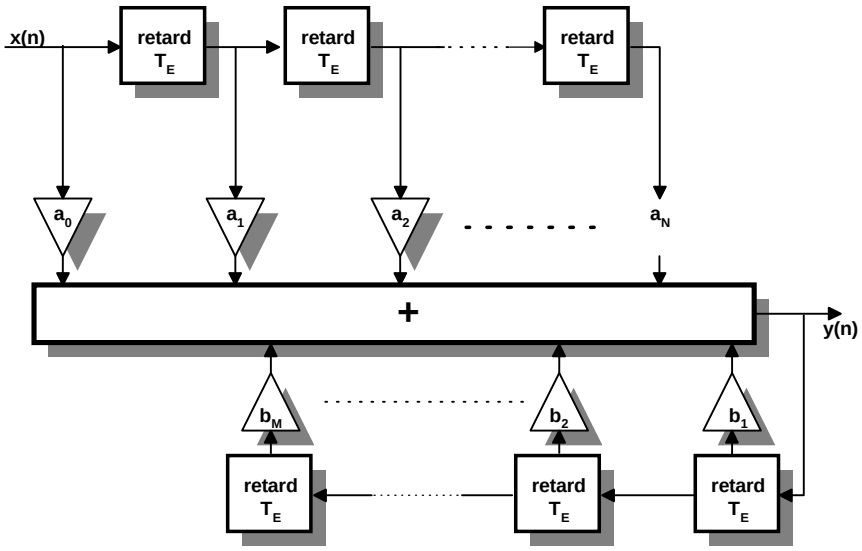


Le système numérique ne pouvant stocker qu'un nombre fini d'échantillons, ce type de filtre est dit à réponse impulsionnelle finie (**RIF ou FIR :Finite Impulse Reponse**). Il est inconditionnellement stable (pas de rebouclage de la sortie), mais présente des performances réduites.

Filtres récursifs

Il est possible d'améliorer les performances du filtre (raideur de la coupure accrue pour un nombre de coefficients plus faible) en rebouclant la sortie sur l'entrée et implanter l'équation suivante :

$$\begin{aligned} y(n) &= a_0.x(n) + a_1.x(n-1) + a_2.x(n-3) + \ldots\ldots + b_1.y(n-1) + b_2.y(n-2) + \ldots\ldots \\ &= \sum_{k=0}^{k=N} a_k.x(n-k) + \sum_{k=1}^{k=M} b_k.y(n-k) \end{aligned}$$



La réponse impulsionnelle de ces filtres est généralement infinie (mais ce n'est pas toujours le cas) et on parle de filtre **RII ou IIR (Infinite Impulse Reponse)**. Ils peuvent être instables, soit parce qu'ils ont été calculés pour (générateur DTMF par exemple) ou à cause des troncatures et arrondis de calculs.

Transformée en z

Comme on peut le voir, l'équation récurrente n'est pas toujours très facile à manipuler, aussi est-il souvent préférable d'introduire une transformation, dite en « z », dont l'opérateur z^{-1} correspond à un retard d'une période T_E de l'horloge d'échantillonnage ; z^{-n} correspondra alors à un retard de n période d'horloge.

L'équation récurrente d'un filtre récursif quelconque :

$$y(n) = a_0.x(n) + a_1.x(n-1) + a_2.x(n-3) + ... + b_1.y(n-1) + b_2.y(n-2) + ...$$
$$= \sum_{k=0}^{k=N} a_k.x(n-k) + \sum_{k=1}^{k=M} b_k.y(n-k)$$

devient alors :

$$Y(z) = a_0.X(z) + z^{-1}.a_1.X(z) + z^{-2}.a_2.X(z) + ... + z^{-1}.b_1.Y(z) + z^{-2}.b_2.Y(z) + ...$$
$$= \sum_{k=0}^{k=N} z^{-k}.a_k.X(z). + \sum_{k=1}^{k=M} z^{-k}.b_k.Y(z)$$

ou encore :

$$H(z) = \frac{Y(z)}{X(z)} = \frac{a_0 + a_1.z^{-1} + a_2.z^{-2} + ...}{1 + b_1.z^{-1} + b_2.z^{-2} + ...} = \frac{\sum_{k=0}^{k=N} z^{-k}.a_k}{1 + \sum_{k=1}^{k=M} z^{-k}.b_k}$$

H(z) représentant la transformée en z du filtre numérique et permettra par exemple l'étude de la stabilité.

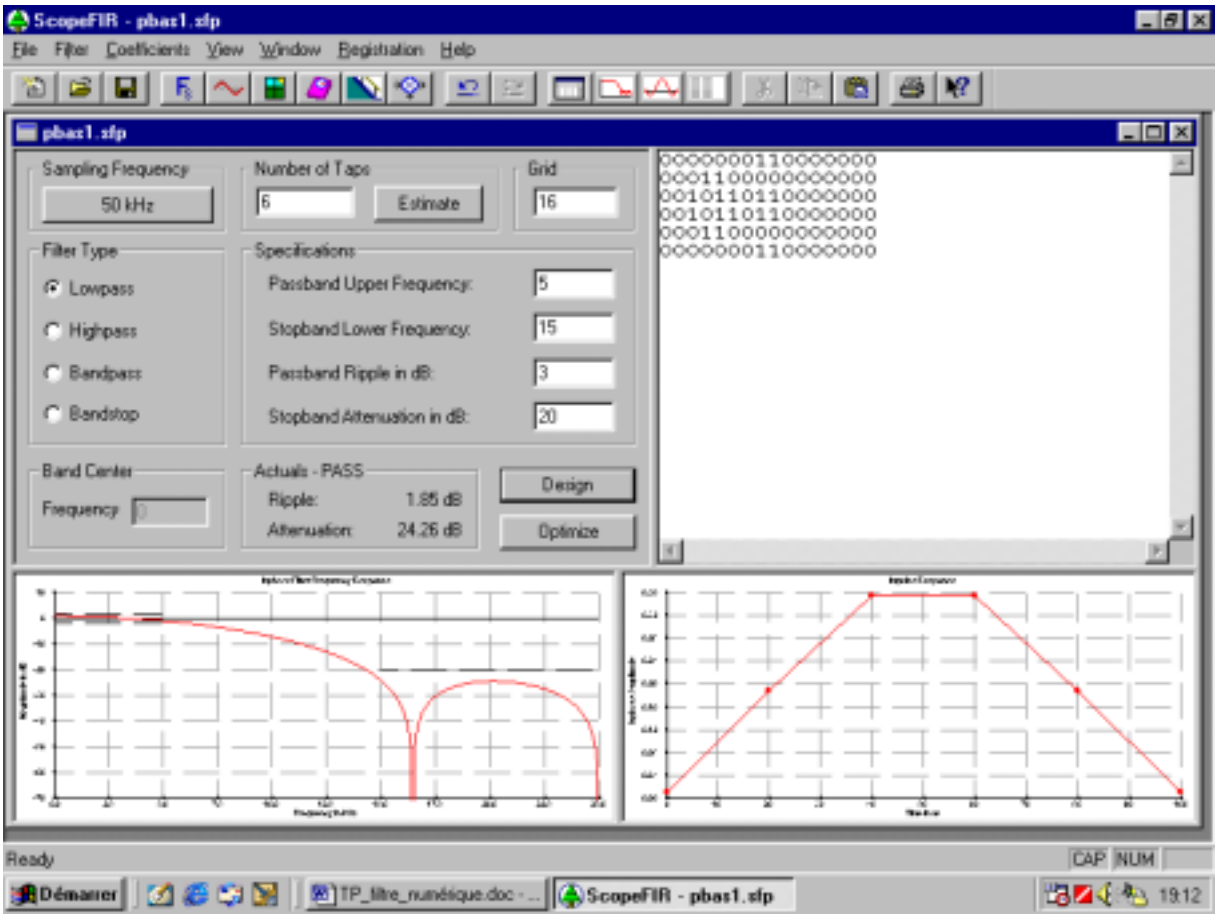
Réponse fréquentielle

L'opérateur z^{-k} représente un retard de k coups d'horloge ; en le remplaçant par l'opérateur retard équivalent de la transformée de Fourrier, à savoir $e^{-j\omega k T_E}$, on obtient la réponse fréquentielle du filtre.

$$H(j\omega) = \frac{\sum_{k=0}^{k=N} e^{-j\omega k T_E}.a_k}{1 + \sum_{k=1}^{k=M} e^{-j\omega k T_E}.b_k}$$

Annexe 2 : utilisation de ScopeFIR

ScopeFIR est un logiciel de calcul des coefficients d'un filtre numérique à réponse impulsionnelle finie ; il est disponible gratuitement sur internet en version d'évaluation.
Son fonctionnement est basé sur l'algorithme de calcul dit de Parks-McClellan. A l'ouverture du logiciel, demander dans un premier temps la version simplifiée du calcul. La fenêtre suivante s'ouvre alors :



On précisera :

- la fréquence d'échantillonnage ;
- le type de filtre (passe bas, passe haut etc…) ;
- les paramètres du gabarit ;
- le nombre de coefficients (Number of Taps) ; celui-ci peut être estimé en fonction des paramètres précédents (Estimate) ;
- éventuellement la grille (paramètre inhérent à l'algorithme de calcul) si le logiciel est dans l'impossibilité d'effectuer le calcul ;
- le nombre de bit de quantification (icône de l'escalier) ;

En appuyant sur le bouton Design, s'affiche alors le gain (on peut changer pour la phase ou le temps de propagation dans le menu **View / Frequency Reponse**), la réponse impulsionnelle, et les coefficients du filtre. Le format de ces derniers (binaire, hexadécimal etc…) peut être changé par le menu contextuel de la fenêtre d'affichage (**Select Filter and Data Format**).

Annexe 3 : Codage des nombres binaires

Représentation

Trois codes sont particulièrement utilisés pour la représentation des nombres binaire :

- le binaire naturel (BN) ; c'est le code le plus simple, et c'est celui utilisé si les nombres restent positifs.
- le binaire décalé (BD) ; c'est le code le plus simple à mettre en œuvre sur les CAN et CNA si des nombres négatifs doivent être représentés ; comme son nom l'indique, il présent simplement un décalage de la moitié de la dynamique par rapport au binaire naturel, afin de représenter des nombres négatifs. Le MSB alors le bit de signe et est au NL1 pour les nombres positifs.
- le complément à 2 (CP2) ; c'est le code qui permet de faire des calculs signés ; les valeurs positives sont codées de la même manière qu'en BN, et les valeurs négatives sont obtenues en complémentant et en ajoutant 1. Le MSB est alors le bit de signe et est au NL1 pour les nombres négatifs.

Le tableau suivant donne un exemple des valeurs décimales obtenues sur 3 bits, suivant l'interprétation du code :

représentation binaire	valeur décimale en BN	valeur décimale en BD	valeur décimale en CP2
111	7	3	-1
110	6	2	-2
101	5	1	-3
100	4	0	-4
011	3	-1	3
010	2	-2	2
001	1	-3	1
000	0	-4	0

On peut noter que l'on passe du CP2 au BD et réciproquement, simplement en complémentant le MSB.

L'augmentation du nombre de bits se traduit :

- en BN par l'ajout de zéros à gauche ;
- en BD par le décalage à gauche du MSB et son remplacement dans les espaces intermédiaires laissés vide par des 0 ;
- en CP2 par le décalage à gauche du MSB et son remplacement dans les espaces intermédiaires laissés vide par la valeur du MSB.

Opération sur des entiers

Additions

L'addition de deux nombres sur B bits donne un nombre sur B+1 bits, que la représentation soit signée (CP2) ou non (BN) ;
exemple en BN sur 3 bits : 7+7=14, il faut 4 bits pour représenter 14 ($14_D=1110_{BN}$) ;
exemple en CP2 sur 3 bits –4-4= - 8, il faut 4 bits pour représenter – 8 (- $8_D=1000_{CP2}$) .

Multiplications

La multiplication de deux nombres sur B bits donne un nombre sur 2B bits, que la représentation soit signée (CP2) ou non (BN) ;
exemple en BN sur 3 bits : 7x7=49, il faut 6 bits pour représenter 49 ($49_D=11\ 0001_{BN}$) ;
exemple en CP2 sur 3 bits 4x4=16, il faut 4 bits pour représenter 16 ($16_D=01\ 0000_{CP2}$, le MSB devant être à 0 pour un nombre positif).

Représentation de nombres réels

Le calcul des filtres nécessite l'utilisation de nombres réels signés ; deux représentations sont envisageables :

- virgule flottante
- virgule fixe.

Nous ne nous intéresserons qu'à ce dernier cas.
Le principe de base reste le complément à 2, mais sur un mot de B bits :

- 1 bit (le MSB) sera utilisé pour représenter le signe,
- B-k-1 pour représenter la partie entière en somme de puissances positives de 2 (2^0, 2^1, 2^2…).
- k bits seront utilisés pour représenter la partie décimale en somme de puissances négatives de 2 (2^{-1}, 2^{-2}, 2^{-3}…).

Par exemple, sur 8 bits, si 3 bits sont destinés à la partie décimale, le nombre 0110 1111 représentera :
$+(2^3+ 2^2+ 2^1+ 2^{-1}+ 2^{-2}+ 2^{-3})=13,875$
Dans le cas où le MSB est au niveau logique 1, on utilise la méthode classique du CP2.
Ce codage serait appelé Q.3 chez Texas Instrument et 5.3 chez Analog Devices.

La gestion de l'extension de bits lors des opérations se fait de même manière que lors du calcul avec des entiers.

On s'arrange souvent pour travailler avec des nombres de valeur absolue maximale inférieure à l'unité, de manière à ne pas avoir à gérer les dépassements dans le cas des multiplications. Le format sur 16 bits par exemples est alors le Q.15 (chez T.I. ou 1.15 chez A.D.).

Annexe 4 : exemples de solutions

Convertisseur BD – CP2 et CP2 – BD

```
library ieee;
use ieee.std_logic_1164.all;

entity BD_CP2 is
generic (N : integer :=8);
port	(	DOUT		: out std_logic_vector (N-1 downto 0);
		DIN		: in std_logic_vector (N-1 downto 0));
end BD_CP2;

architecture arch of BD_CP2 is
begin
DOUT(N-2 downto 0)<=DIN(N-2 downto 0);
DOUT(N-1)<= not DIN(N-1);
END arch;
```

Additionneur deux entrées

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
--les paquetages std_logic_1164 et std_logic_unsigned de la bibliothèque ieee
--permettent respectivement l'utilisation du type std_logic et l'addition avec ce type

entity ADD_SGN is
-- additionneur en complément à 2
-- nombres d'entrée sur N bits, résultat sur N+1 bits
generic (	N : integer:=8);
port	(	A, B		: in std_logic_vector (N-1 downto 0);
		R		: out std_logic_vector (N downto 0));
end ADD_SGN;

architecture arch of ADD_SGN is
begin
R<=A+B;
END arch;
```

Additionneur à 3 entrées

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
--les paquetages std_logic_1164 et std_logic_unsigned de la bibliothèque ieee
--permettent respectivement l'utilisation du type std_logic et l'addition avec ce type

entity ADD_3_SGN is
-- additionneur 3 entrée en complément à 2
-- nombres d'entrée sur N bits, résultat sur N+2 bits
generic (	N : integer:=3);
port	(	A, B, C		: in std_logic_vector (N-1 downto 0);
			R		: out std_logic_vector (N+1 downto 0));
end ADD_3_SGN;

architecture arch of ADD_3_SGN is
```

```
signal XA : std_logic_vector (N+1 downto 0);
signal XB : std_logic_vector (N+1 downto 0);
signal XC : std_logic_vector (N+1 downto 0);

begin
XA(N+1)<=A(N-1);    XA(N)<=A(N-1);    XA(N-1 downto 0)<=A(N-1 downto 0);
XB(N+1)<=B(N-1);    XB(N)<=B(N-1);    XB(N-1 downto 0)<=B(N-1 downto 0);
XC(N+1)<=C(N-1);    XC(N)<=C(N-1);    XC(N-1 downto 0)<=C(N-1 downto 0);

R<=XA+XB+XC;

END arch;
```