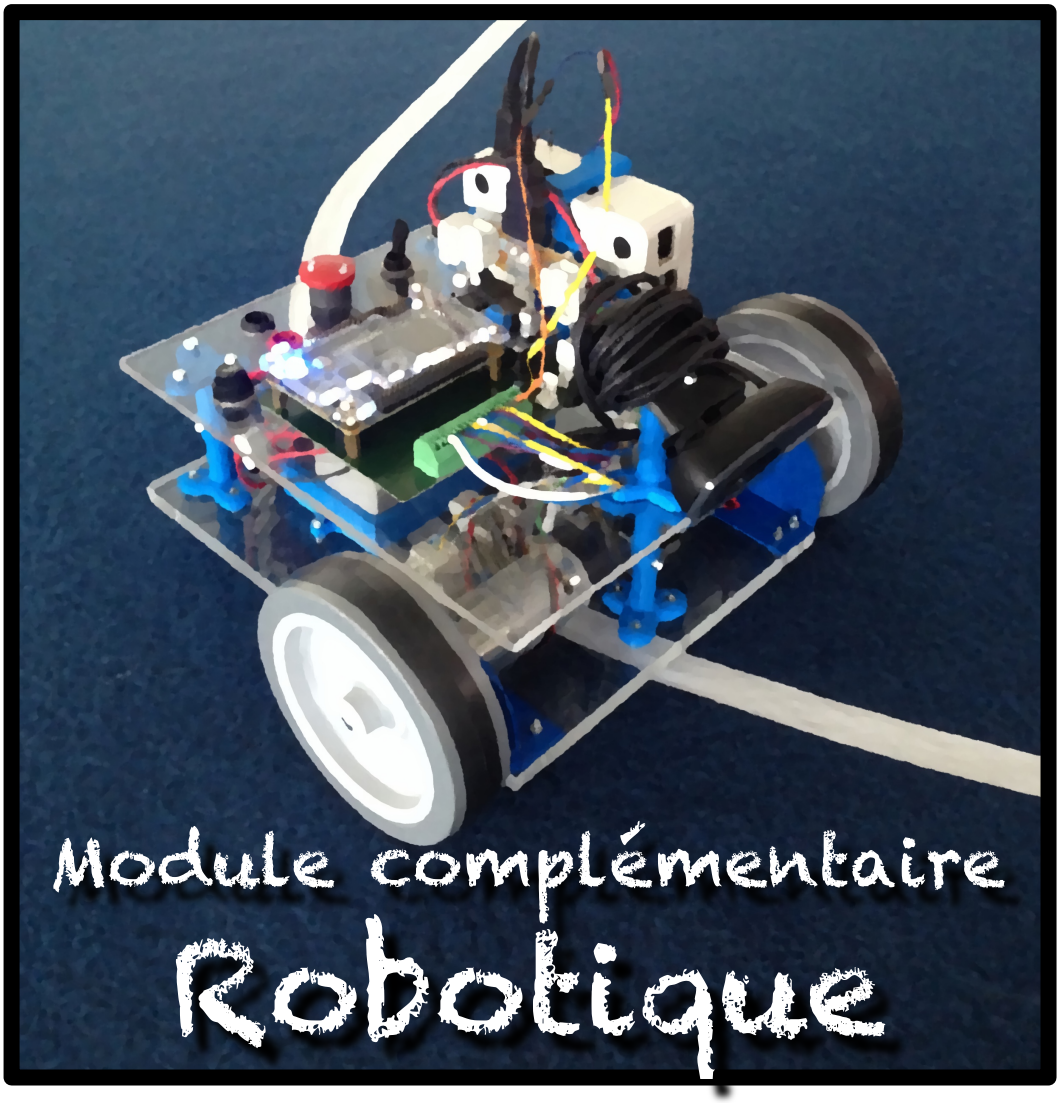


Filtrage numérique et implémentation sur FPGA

Sommaire

1. Établissement de l'architecture du filtre

Exemple : On cherche à implémenter sur un FPGA, un filtre du second ordre dont la transmittance est donnée ci-dessous :

$$C(p) = K\frac{1+Tn.p}{1+2.m.T_0.p+T_0{}^2.p^2} = \frac{Ns}{Ne}$$

1.1. Étape 1 : Exprimer la relation entre l'entrée et la sortie

$$Ns + 2m.T_0.p.Ns + T_0{}^2.p^2.Ns = K.Ne + K.Tn.p.Ne$$

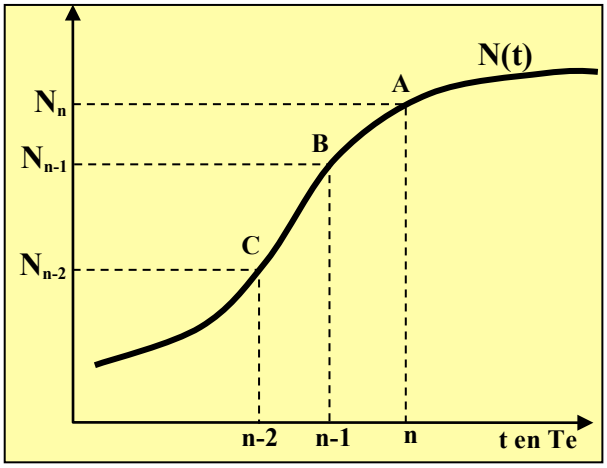
1.2. Étape 2 : Passer dans le domaine temporel

Rappel : La multiplication par ***p*** dans le domaine de Laplace est équivalente à la dérivation par rapport au temps dans le domaine temporel.

$$Ns + 2m.T_0.\frac{d}{dt}(Ns) + T_0{}^2.\frac{d^2}{dt^2}(Ns) = K.Ne + K.Tn.\frac{d}{dt}(Ne)$$

1.3. Étape 3 : Passer en échantillonné

Le Filtre étant réalisé par un composant numérique (FPGA), il ne sera pas possible de suivre l'évolution du signal d'entrée (ni d'imposer le signal de sortie) de manière continue. On travaille donc avec des signaux échantillonnés, on ne connaît donc la valeur des signaux que de manière périodique toutes les T_e secondes. (T_e étant la période d'échantillonnage).



Approximation des dérivées successives :

Si la période d'échantillonnage T_e est suffisamment petite par rapport à la dynamique d'évolution du signal d'entrée, le signal va évoluer très peu entre deux instants d'échantillonnage. On peut donc faire l'approximation de la dérivée première par la pente de la corde :

$$\frac{d}{dt}(Nv) \approx \frac{Nv_n - Nv_{n-1}}{Te}$$

et celle de la dérivée seconde :

$$\frac{d^2}{dt^2}(Nv) = \frac{Nv_n - 2Nv_{n-1} + Nv_{n-2}}{Te^2}$$

D'ou la relation suivante :

$$Ns_n + 2m.T_0.\frac{Ns_n - Ns_{n-1}}{Te} + T_0{}^2.\frac{Ns_n - 2Ns_{n-1} + Ns_{n-2}}{Te^2} = K.Ne_n + K.Tn.\frac{Ne_n - Ne_{n-1}}{Te}$$

1.4. Étape 4 : exprimer Ns_n en fonction des autres échantillons

$$Ns_n(1+\frac{2m.T_0}{Te}+\frac{T_0{}^2}{Te^2}) = Ne_n(K+\frac{K.Tn}{Te}) - Ne_{n-1}(\frac{K.Tn}{Te}) + Ns_{n-1}(\frac{2m.T_0}{Te}+\frac{2.T_0{}^2}{Te^2}) - Ns_{n-2}\frac{T_0{}^2}{Te^2}$$

D'où l'équation de récurrence suivante :

$$Ns_n = Ne_n\frac{K+\frac{K.Tn}{Te}}{1+\frac{2m.T_0}{Te}+\frac{T_0{}^2}{Te^2}} - Ne_{n-1}\frac{\frac{K.Tn}{Te}}{1+\frac{2m.T_0}{Te}+\frac{T_0{}^2}{Te^2}} + Ns_{n-1}\frac{\frac{2m.T_0}{Te}+\frac{2.T_0{}^2}{Te^2}}{1+\frac{2m.T_0}{Te}+\frac{T_0{}^2}{Te^2}} - Ns_{n-2}\frac{\frac{T_0{}^2}{Te^2}}{1+\frac{2m.T_0}{Te}+\frac{T_0{}^2}{Te^2}}$$

Finalement on obtient :

$$Ns_n = a_0.Ne_n + a_1.Ne_{n-1} + b_1.Ns_{n-1} + b_2.Ns_{n-2}$$

Avec :

$$a_0 = \frac{K + \frac{K.Tn}{Te}}{1 + \frac{2m.T_0}{Te} + \frac{T_0^2}{Te^2}} \quad a_1 = -\frac{\frac{K.Tn}{Te}}{1 + \frac{2m.T_0}{Te} + \frac{T_0^2}{Te^2}} \quad b_1 = \frac{\frac{2m.T_0}{Te} + \frac{2.T_0^2}{Te^2}}{1 + \frac{2m.T_0}{Te} + \frac{T_0^2}{Te^2}} \quad b_2 = -\frac{\frac{T_0^2}{Te^2}}{1 + \frac{2m.T_0}{Te} + \frac{T_0^2}{Te^2}}$$

Application numérique : K = 10 ; Tn = 120 ms ; T_0 = 50ms ; m = 0,7 ; Te = 10 ms

$$a_0 = \frac{130}{33} \quad a_1 = -\frac{120}{33} \quad b_1 = \frac{57}{33} \quad b_2 = -\frac{25}{33}$$

1.5. Architecture du filtre

Les symboles utilisés		
Multiplieur	Additionneur	Bloc retard
α	+	Z^{-1}

Équation récurrente :

$$Ns_n = a_0.Ne_n + a_1.Ne_{n-1} + b_1.Ns_{n-1} + b_2.Ns_{n-2}$$

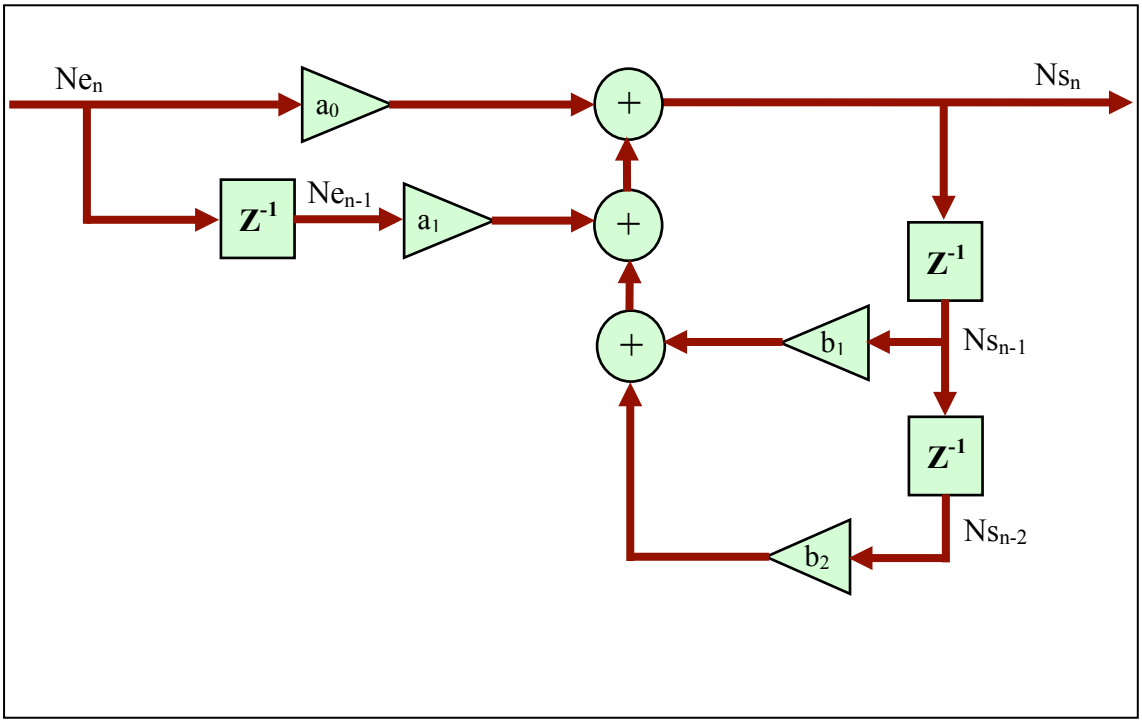
Nota : a_1 et b_2 sont négatifs

Explications :

Le multiplieur réalise la multiplication du signal par un coefficient réel signé.

L'additionneur permet d'additionner deux signaux signés.

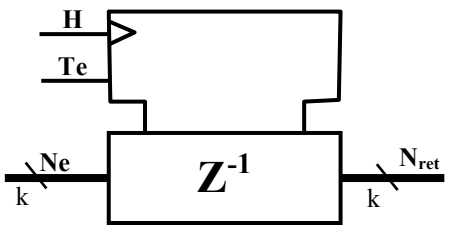
Le bloc retard effectue une mémorisation qui permet de retenir l'état du signal à l'échantillon précédent.



2. Réalisation des différents blocs du filtre

2.1. Bloc retard

Il s'agit de réaliser un composant qui mémorise la valeur d'entrée pendant une période d'échantillonnage.




```
architecture a_Z_moins_1 of Z_moins_1 is
begin
   process(H)
   begin
      if (H'event and H='1') then
         if Te='1' then
            Nret<=Ne;
         end if;
      end if;
   end process;
end;
```

2.2. Multiplieur

2.2.1. Format des nombres

➢ **multiplication de 2 nombres non signés**

Exemple : 15 *7

Nombre de bits du produit : 4 bits * 3bits ➔ 7 bits
dans le cas général : n bits * p bits ➔ n+p bits

			1	1	1	1		15
		*		1	1	1		* 7
1	10	10	10	1	0			
			1	1	1	1		
		1	1	1	1	0		
	1	1	1	1	0	0		
1	1	0	1	0	0	1		105

➢ **multiplication de 2 nombres signés en complément à 2**

Exemple (-16) *(-8)

Nombre de bits du produit : 5 bits * 4bits ➔ 9 bits

				1	0	0	0	0	-16
				*	1	0	0	0	* -8
0	1	0	0	0	0	0	0	0	+128

Conclusion : que les nombres soient signés ou non signés, le produit d'un nombre codé sur n bits et d'un nombre codé sur p bits s'exprimera sur n+p bits

2.2.2. Multiplication par un coefficient non entier (nombre binaire à virgule)

Exemple : le coefficient multiplicateur est : +/- 95/14 précision 1/100

Rappel : Codage d'un entier à virgule fixe

➢ *Détermination du format :*
- ***Partie entière*** : 95/14 = 6,... ➔ pour coder 6 il faut 3 bits : 110 plus un bit de signe. La partie entière s'écrira donc avec un minimum de 4 bits : 95/14 = 0110,….
- ***Mantisse*** : Pour une précision de 1/100 il faut prendre 7 bits après la virgule. Le poids du quantum est alors : 1/128 < 1/100 précision voulue.

➢ *Expression du coefficient en quanta :*
Le quantum vaut 1/128. Donc le coefficient multiplicateur exprimé en quanta sera : Coef_q = 95/14*128= 868,57. Ce coefficient ne peut être qu'entier. Il faut donc arrondir à l'entier le plus proche : Coef_q = 869

➢ *Conversion en binaire naturel :* Coef_q = 869 = $(365)_{16}$ = $(11\ 0110\ 0101)_2$

➢ *Placement du bit de signe :*
Pour Coef = +95/14 ➔ le bit de signe est 0 qu'on ajoute aux bits déjà existants ➔ **Coef_q =011 0110 0101**
Pour Coef = -95/14 ➔ il faut prendre l'opposé (complément à 2) du nombre positif.
C2(N) = C1(N) + 1 ➔ on inverse tous les bits : C1(N) =10010011010 puis on ajoute 1: **C2(N) =10010011011.**

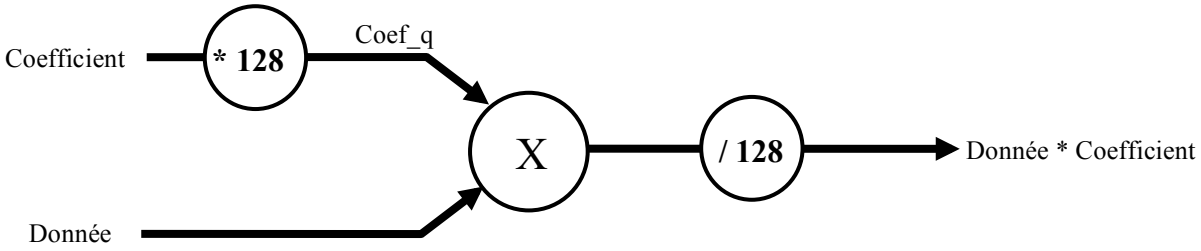
➢ *Placement de la virgule :*
Quand le coefficient est positif la partie entière doit être 6
Pour le coefficient négatif le bit de poids fort doit être obligatoirement un 1

Indice du bit	5	4	3	2	1	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	
Valeur du bit			**1**	**0**	**0**	**1,**	**0**	**0**	**1**	**1**	**0**	**1**	**1**				
Poids du bit	**32**	**16**	**8**	**4**	**2**	**1**	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$	$\frac{1}{32}$	$\frac{1}{64}$	$\frac{1}{128}$	$\frac{1}{256}$	$\frac{1}{512}$	$\frac{1}{1024}$	

➢ Remarques :
- si l'on veut prendre un bit de plus pour la partie entière il faut rajouter devant un 0 si le nombre est positif et un 1 si le nombre est négatif.
- Placer la virgule revient à diviser par 2^7 = 128.

2.2.3. Principe de la multiplication

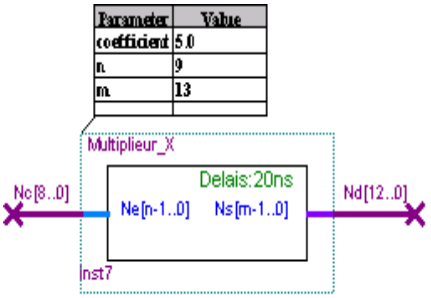
Pour l'exemple pris (précision > 1/100) on réalise les opérations suivantes



2.2.4. Paramètre réel

Plutôt que de calculer à la main le coefficient en binaire à virgule fixe on veut déclarer le coefficient multiplicateur comme un paramètre réel et laisser au compilateur la conversion. Voici les paramètres que nous allons utiliser.

```
Generic(
   Coef:real:=-15.99; --coefficient multiplicateur
   v:integer:=10;     --nombre de bits de la mantisse<=>précision : v=10: précision=1/1000
   n:integer:=9;      --largeur du bus d'entrée
   m:integer:=13      --largeur du bus de sortie. m >= n+ nombre minimum
   );  -- on doit avoir m-n+1 >= nombre de bits de la partie entière du coefficient, bit de signe compris
```



2.2.5. Architecture du multiplieur

```
ARCHITECTURE a_Multiplieur_X OF Multiplieur_X IS
   --signal CoefQ:integer range -2**v*(integer(abs(coefficient))) to 2**v*(integer(abs(coefficient)));
   signal CoefQ:integer range -2**(m-n+v) to 2**(m-n+v)-1; --Déclaration du signal interne Coef_Q
   signal temp:integer range -2**(m+v-1) to 2**(m+v-1)-1;
BEGIN
   CoefQ<=integer(round(Coef*(2**real(v)))); --Calcul du coefficient en quantum.
   temp<=Ne*CoefQ;                  --temp: registre temporaire suffisamment grand
   No<=temp/(2**v);
END;
```

- **integer(Nombre_Réel)** ➔ convertit un nombre réel en nombre entier : si le nombre réel a des décimales celles ci seront perdues
- **real(Nombre_Entier)** ➔ convertit un nombre entier en un nombre réel de même valeur. (si l'argument est 2 l'instruction retourne 2.0)
- **round(Nombre_Réel)** ➔ arrondit le nombre réel au nombre entier le plus proche. L'instruction retourne un réel.
- **2**(Nombre_Entier)** ➔ élève 2 à la puissance « Nombre_Entier ».
- **Coef*(Nombre_Réel)** ➔ Multiplie 2 nombres réels. L'instruction retourne un réel.
- **abs(coefficient)** ➔ Prend la valeur absolue du paramètre. L'instruction retourne un réel si l'argument est un réel et un entier si l'argument est un entier.

Remarques :

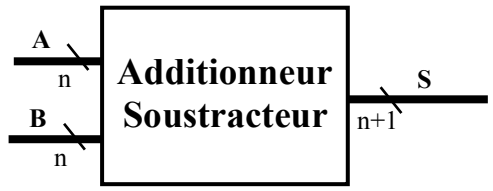
- l'introduction du registre temporaire « **temp** » n'est pas obligatoire on peut très bien s'en passer et écrire directement l'instruction : **No<=Ne*CoefQ/(2**v);**
- L'ordre des opérations est important : **No<=Ne*CoefQ/(2**v);** est différent de **No<=Ne/(2**v)*CoefQ;**
  - ✓ Donnez les valeurs de ces expressions pour **Ne = 75, v = 7,** et **CoefQ = 128**.

    Ne*CoefQ/(2v) = (75*128)/128 = 9600/128 = 75**

    Ne/(2v)*CoefQ = (75/128)*128 = 0*128 = 0**

- Lors de l'instanciation, le paramètre va être changé selon les besoins. Si on désire un coefficient de 5 il faudra écrire : 5.0 sinon en déclenche une erreur en VHDL. En effet 5 est un entier, or le paramètre « Coefficient » a été déclaré en réel.

2.3. Additionneur-soustracteur sans perte de précision

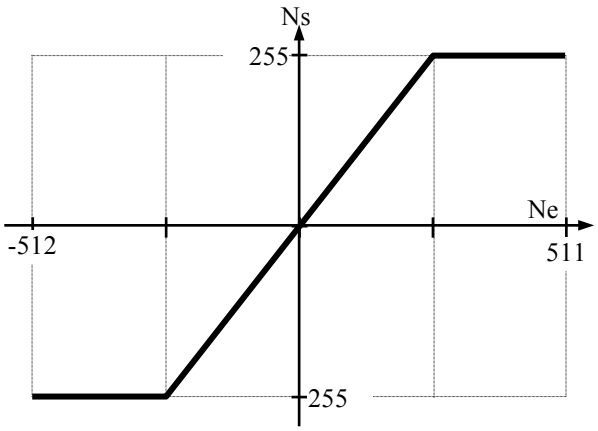


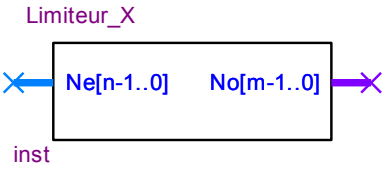

```
-- Additionneur soustracteur paramétrique
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
use ieee.std_logic_signed.all;

ENTITY additionneur_Soustracteur_X IS
generic
   (
   Addition:boolean:=true;
   n:integer:=9
   );
PORT
    (
   A,B:IN std_logic_vector(n-1 downto 0);
   S:out std_logic_vector(n downto 0)
   );
END;

 ARCHITECTURE a_additionneur_Soustracteur_X 0F
                          additionneur_Soustracteur_X IS
   signal Ai,Bi:std_logic_vector(n downto 0);
BEGIN
   Ai<= '0'& A when A>=0 else '1'& A;
   Bi<='0'& B when B>=0 else '1'& B;
   S<=Ai+Bi when Addition else Ai-Bi;
END;
```

<u>Exercice</u> : Réalisez un additionneur générique qui additionne 4 entrées de n bits ;

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
use ieee.std_logic_signed.all;

ENTITY additionneur4_X IS
generic
   (
   n:integer:=9
   );
PORT
    (
   A,B,C,D:IN std_logic_vector(n-1 downto 0);
   S:out std_logic_vector(n+1 downto 0)
   );
END;
ARCHITECTURE a_additionneur4_X 0F additionneur4_X IS
 Signal Ai,Bi,Ci,Di:std_logic_vector(n+1 downto 0);
BEGIN
   Ai<=A(n-1) & A(n-1) & A;
   Bi<= B(n-1) & B(n-1) & B;
   Ci<= C(n-1) & C(n-1) & C;
   Di<= D(n-1) & D(n-1) & D;
   S<= Ai+Bi+Ci+Di;
END;
```

2.4. Introduction d'une saturation






```
--Ramène de n bits à m bits
--introduit, si besoin, une saturation
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY Limiteur_X IS
generic(
   n:integer:=13;
   m:integer:=9
   );
PORT(
   Ne:IN  integer range -2**(n-1)to 2**(n-1)-1;
   Ns:out integer range -2**(m-1) to 2**(m-1)-1
   );
END;
ARCHITECTURE a_Limiteur_X OF Limiteur_X IS
BEGIN
   process(Ne)
   begin
      if Ne >2**(m-1)-1 then Ns<=2**(m-1)-1;
      elsif Ne <-2**(m-1) then Ns<=-2**(m-1);
      else Ns<=Ne ;
      end if;
   end process;
END;
```