TD ALM HARD

Le Multiplieur (document enseignant)

1- Opérandes en base 2

Comment calculer 6*5 ?

Pour l'instant entiers naturels

Exemple en base 2 de multiplication :

```
        110
*       101
_____________
        1 1 0
      0 0 0 .
    1 1 0 . .
_____________
    1 1 1 1 0
```

Notons en binaire A=a2,a1,a0 le multiplicande et B= b2,b1,b0 le multiplieur

S=s4,s3,s2,s1,s0

Combien de bits nécessaires pour coder le résultat ? si A et B sur n bits.

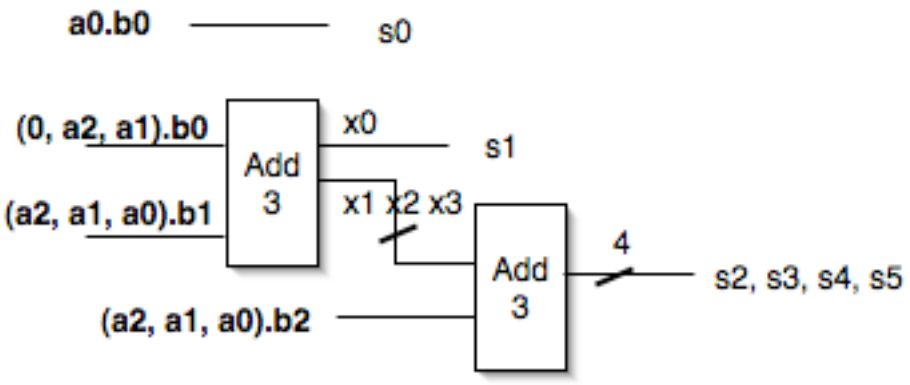
Analyse des opérations effectuées : Et booléen, additions et décalages

Réalisation à l'aide d'additionneurs.

Taille des additionneurs ?

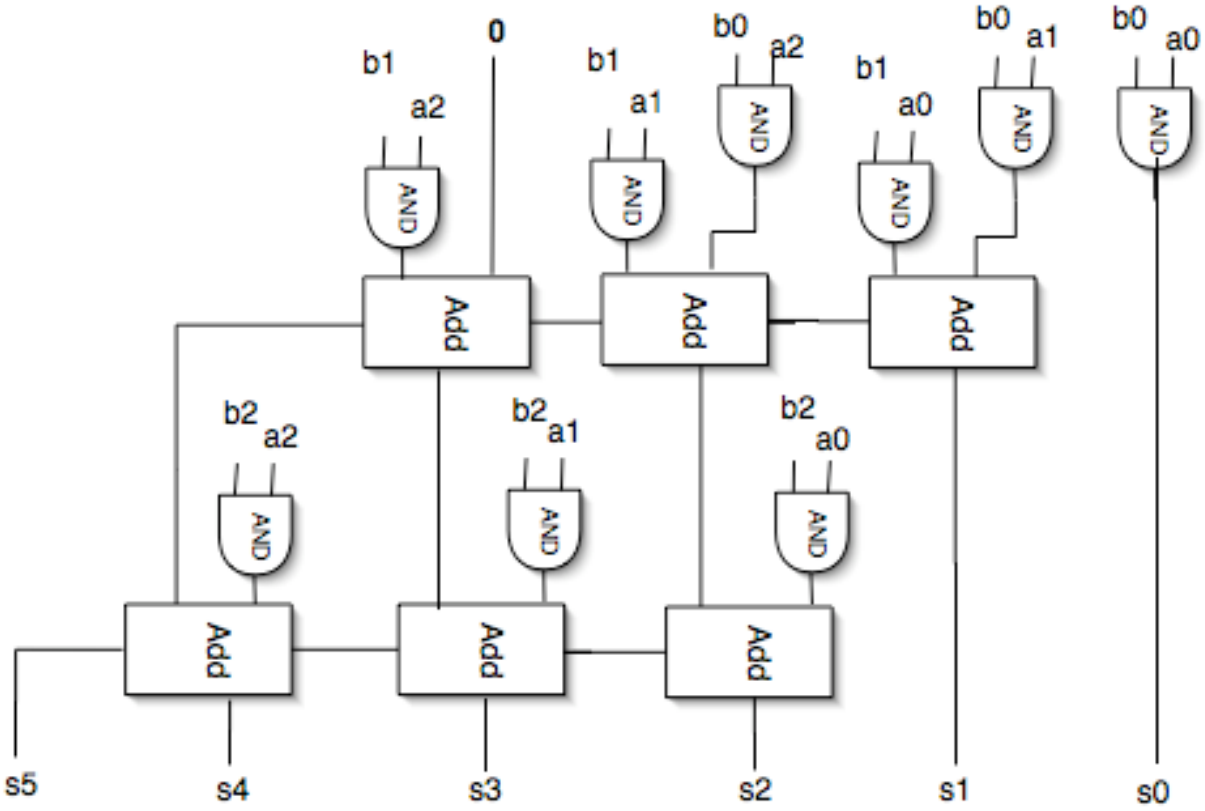
```
        111                         a2 a1 a0
*       111                 *       b2 b1 b0
_____________               _____________
    0 1 1 1                     0   b0.(a2 , a1 , a0)
+   1 1 1 .             +     b1.(a2 , a1 , a0)    0
_____________               _____________________
  1 0 1 0 1               x3    x2    x1    x0    b0.a0
  1 1 1 . .         +   b2.(a2 , a1 , a0)    0     0
_____________     _____________________________
1 1 0 0 0 1         s5  s4    s3    s2    s1    s0
```

Dessin à l'aide d'additionneurs 3 bits :



Si on revient à l'additionneur 1 bits, on arrive au circuit régulier suivant (avec n=3):

```
        111                          a2 a1 a0
*       111                  *       b2 b1 b0
_____________                _____________
    0 1 1 1                      0   b0.a2 b0.a1  b0.a0
+   1 1 1 .              +     b1.a2 b1.a1 b1.a0   0
_____________                _____________________
  1 0 1 0 1                x3   x2   x1    x0     b0.a0
  1 1 1 . .      +       b2.a2 b2.a1 b2.a0  0      0
_____________            _____________________________
1 1 0 0 0 1              s5  s4   s3   s2   s1     s0
```



2- Et si le multiplicande est négatif ?

Le multiplieur est en base 2, le multiplicande est en complément à 2, le résultat aussi.

Il faut propager le bit de poids fort si il est à 1 dans l'étage suivant pour garder des sommes intermédiaires négatives (et cela seulement si le multiplicande est négatif)

Exemple :

```
        111                          a2 a1 a0
*       101                  *       b2 b1 b0
_____________                ____________
    1 1 1 1                  b0.a2  b0.a2 b0.a1  b0.a0
+   0 0 0 .              +   b1.a2 b1.a1 b1.a0    0
_____________                ___________________
  1 1 1 1 1                  x3   x2   x1    x0     b0.a0
  1 1 1 . .      +           b2.a2 b2.a1 b2.a0   0      0
_____________                ___________________________
1 1 1 0 1 1                  s5   s4    s3    s2     s1     s0
```

.

Point de départ : la somme intermédiaire (notée sn … s1) obtenue est juste ou non sur n bits en complément à 2.

Il faut rajouter un bit pour faire la somme suivante (ou pour le résultat final) de façon à que cela soit juste sur n+1 bits.

Dans l'exemple sur 3 bits, on doit rajouter x3 puis s5 (j'oublie le premier cas qui est plus simple).

Il faut donc que la valeur reste juste sur n+1 bits en complément à 2.

Quelle est la valeur de ce bit x que l'on rajoute?

On peut étudier tous les cas pour ce bit à rajouter suivant le flag Overflow V (complément à 2) et la valeur de la dernière retenue sn+1.

Si V =1 alors résultat faux sur n bits mais juste sur n+1 bits (à admettre ou à démontrer) donc x= sn+1

Si V=0 alors le résultat est juste sur n bits il faut donc que x soit égal au signe de la somme(donc sn) pour que la somme reste juste sur n+1 bits.

D'ou x= V.sn+1 + not V. sn

On remplace la formule de V= Ri xor sn+1

On trouve : x=(Ri not sn+1 + not Ri. Sn+1)sn+1 + (Ri.sn+1 + notRi.not Sn+1)sn

X= not Ri . sn+1 + Ri sn+1 sn + notRi.notSn+1 sn

X= not Ri. sn+1 + sn+1 sn + not Ri sn

Autre façon de raisonner en fonction des deux derniers bits des opérandes an et bn.

On remarque ici que les signes des opérandes sont corrélés.

En effet :

-si le premier opérande est négatif (somme précédente non nulle donc multiplicande négatif) alors le deuxième opérande est forcément nul ou négatif.

-de la même façon si le deuxième opérande est négatif (multiplicande négatif) alors forcément le 1er opérande est négative ou nul.

an bn x

0 0 0 si an=0 alors le premier opérande est ≥0. Auquel on ajoute un deuxième opérande ≥0 donc résultat ≥0 donc x=0

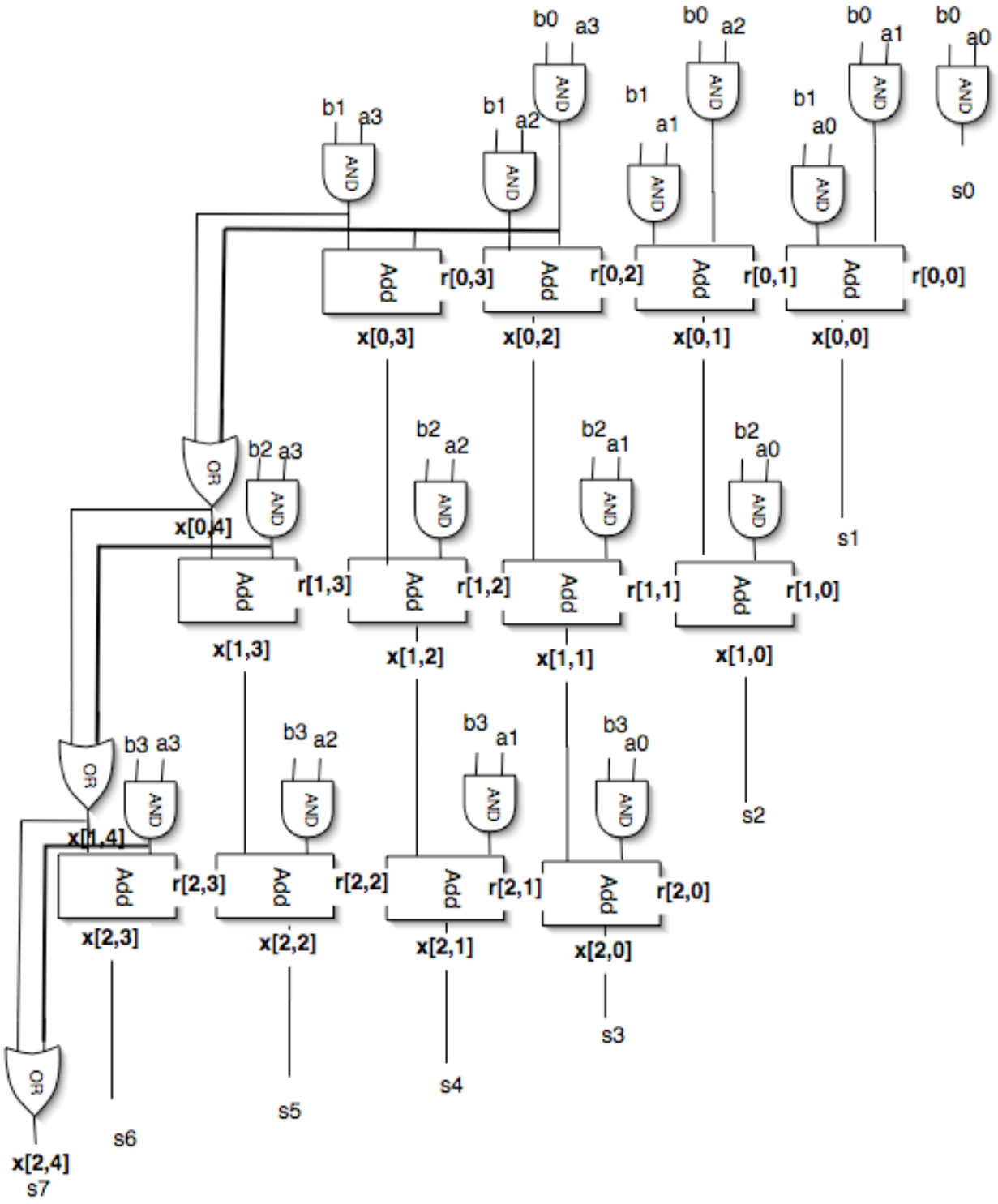
0 1 1 cas ou le deuxième opérande est négatif (multiplicande négatif) donc si an=0 forcément le premier opérande est nul (voir remarque) donc le résultat est négatif donc x=1

1 0 1 somme précédente négative donc 2eme opérande est soit négatif soit nul, donc forcément x=1

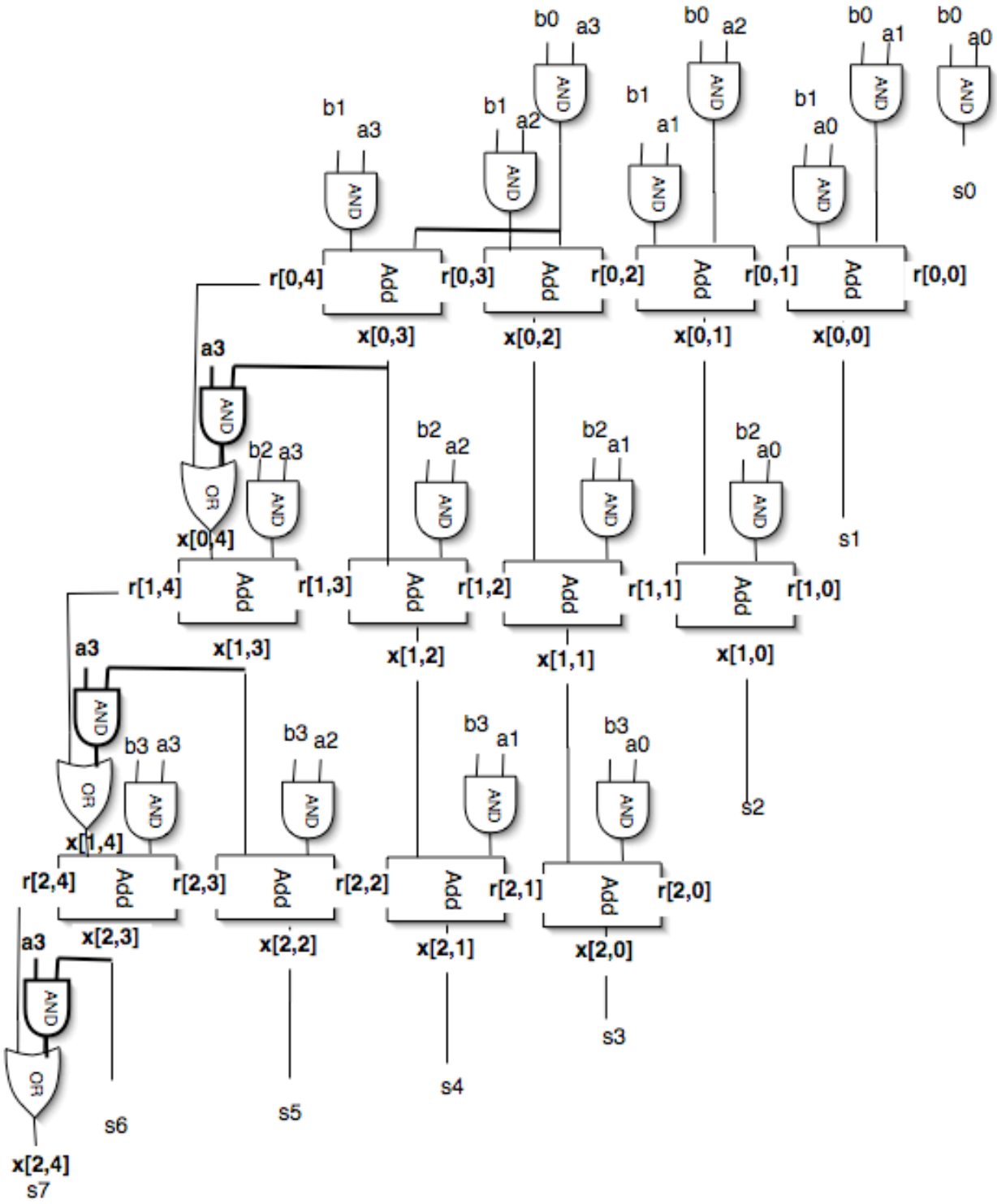
1 1 1 somme précédente négative et 2eme opérande négative donc x=1

D'où x= an + bn

1eme solution : Multiplieur 4 bits avec 1^{ere} opérande et résultat en complément à 2 : x= an+bn


b0 a3
b0 a2
b0 a1
b0 a0
b1 a3
b1 a2
b1 a1
b1 a0
AND
s0
Add
r[0,3]
r[0,2]
r[0,1]
r[0,0]
x[0,3]
x[0,2]
x[0,1]
x[0,0]
OR
b2 a3
b2 a2
b2 a1
b2 a0
x[0,4]
s1
r[1,3]
r[1,2]
r[1,1]
r[1,0]
x[1,3]
x[1,2]
x[1,1]
x[1,0]
b3 a3
b3 a2
b3 a1
b3 a0
x[1,4]
s2
r[2,3]
r[2,2]
r[2,1]
r[2,0]
x[2,3]
x[2,2]
x[2,1]
x[2,0]
s3
s4
s5
s6
x[2,4]
s7

2eme solution : Multiplieur 4 bits avec 1ere opérande et résultat en complément à 2 :
x = signe.sn + sn+1
Solution qui semble juste mais que l'on a du mal à justifier



Pour finir une autre façon de faire :
Une petite réflexion théorique proposée Goran (revue par Pascal) :

Le complément a 2 sur m bits d'un entier positif x sur n bits ($2^{n-1} \leq x < 2^{n-1}$), m>n, est C2(x)= (2^m +x)modulo 2^m (Souvent on prendra m=n+1, c.a.d., on ajoute un bit pour le signe)
L'argument suivant montre que la multiplication binaire est correcte pour les entiers relatifs exprimes en C2 si m>=2n. Formellement, il faut montrer que (C2(x).C2(y))modulo 2^m=C2(x.y) si m>=2n.
Ce qui donne sur m bits

Pour x<0 : C2(x)= 2^m - |x|

Pour x≥0 : C2(x)= x

Etudions les différents cas :

- Si on fait le produit de 2 entiers négatifs :

(2^m-|x|)*(2^m-|y|)= 2^(2m)-(|x|+|y|)*2^m + |x||y|

En regardant sur m bits (c'est à dire modulo 2^m) on obtient (si |x||y| < 2^m) :

|x||y| =x.y= C2(x.y)

- Si on fait le produit d'un entier négatif et d'un positif :

x*(2^m-|y|)= x*2^m - x |y|

En regardant sur m bits (c'est à dire modulo 2^m) on obtient (si –x |y| < 2^m) :

–x |y| = C2(x.y)

- Si on fait le produit de 2 entiers positifs : x*y

En regardant sur m bits (c'est à dire modulo 2^m) on obtient (si x y < 2^m) : x.y= C2(x.y)

En résumé, pour que le calcul soit correct, il suffit d'exprimer x et y en complément à 2 sur m bits avec m>=2n *avant* la multiplication pour x.y < 2^m (on vérifie que m>=2n implique m>n). La multiplication même s'effectue comme pour les entiers naturels. On ne garde que les m bit de poids faible du résultat.

ANNEXE : Description lustre :

Sur 4 bits avec variables intermédiaires pour comprendre :

Réalisation en Lustre (à voir plus tard avec les étudiants quand ils auront appris le Lustre)
Faux générique sur 4 bits en base 2

```
--multiplieur 4 bits non générique realise a partir de and 2 entrees et d'add 1 bit
--2 opérandes n bits, résultats 2*n bits le tout en base 2
------------------------------------------------------------
-- Additionneur 1 bit
node add1bit( x, y, rin : bool)  returns   (s, rout : bool);
let
      s = x xor y xor rin;
      rout = (x and y) or (x and rin) or (y and rin);
tel
------------------------------------------------------------
--------multiplieur 4 bits
--Voir le dessin 1er indice : colonne, 2eme: ligne
node multnbits(const nn: int; A, B: bool^nn) returns (S: bool^(2*nn));
var x0,x1,x2, r0,r1,r2: bool^(nn+1); --resultat et retenues intermediaires
let
--premier etage
r0[0]= false;
(x0[0..nn-2], r0[1..nn-1])= add1bit( B[0]^(nn-1) and A[1..nn-1], B[1]^(nn-1) and A[0..nn-2],
r0[0..nn-2]);
(x0[nn-1],r0[nn])=add1bit(B[1] and A[nn-1], 0, r0[nn-1]);
x0[nn]= r0[nn];
-- etages de 1 a nn-2
r1[0]= false;
r2[0]= false;
x1[nn]=r1[nn];
x2[nn]=r2[nn];
(x1[0..nn-1],r1[1..nn])= add1bit(B[2]^nn and A[0..nn-1], x0[1..nn], r1[0..nn-1]);
```

```
(x2[0..nn-1],r2[1..nn])= add1bit(B[3]^nn and A[0..nn-1], x1[1..nn], r2[0..nn-1]);
--les sorties
S[0]= A[0] and B[0];
S[1]= x0[0];
S[2]= x1[0];
S[nn-1..(2*nn-1)]= x2[0..nn];
tel;

-- instantiation du multnbits
const a=4;

node instmultnbits(op1, op2: bool^a)
returns(res: bool^(2*a));
let
res= multnbits(a, op1, op2);
tel;
```

Vrai générique (plus compliqué tableau de tableau) :

```
--multiplieur n bits realise a partir de and 2 entrees et d'add 1 bit
--2 opérandes n bits, résultats 2*n bits
--le multiplicande est en complément à 2, il peut etre négatif
--le multiplieur est en base 2
--Le resultat est en complement à 2
------------------------------------------------------------
--tableau de tableau m*n
--S[i][j]= A[j]and B[i])
--réalisation recursive

node tabandrec(const n,m : int; A : bool^n; B: bool^m) returns (S: bool^n^m);
--attention tableau de tableau : 1er indice contenu des tableau, 2eme indice :tableau de tableau
--indice inverse dans l'utilisation par rapport à la declaration: ici m tableau de n élements: S[0..m-
1,0..n-1]
```

```
--attention ecriture S[0..m-1][0..n-1] autre semantique

let
-- attention utiliser with au lieu de if sinon le compilateur produit un code qui boucle ??
S=with (m=1) then ([B[0]^n and A])
        else  ( tabandrec(n, m-1, A , B[0..m-2])|[B[m-1]^n and A]);
        --| est l'opération de concatenation de tableau
tel;
-------------------------------------------------------------
--------multiplieur base sur des tableaux de tableaux
--Voir le dessin 1er indice : colonne, 2eme: ligne

node multnbits(const nn: int; A, B: bool^nn)
returns(S: bool^(2*nn));
var r, x: bool^(nn+1)^nn-1; --resultat et retenues intermediaires
taband:bool^nn^(nn-2); --tableau des and ai bj

let
--premier etage---------
r[0,0]= false;
(x[0,0..nn-2], r[0,1..nn-1])= add1bit( B[0]^(nn-1) and A[1..nn-1], B[1]^(nn-1) and A[0..nn-2],
r[0,0..nn-2]);
(x[0, nn-1],r[0, nn])= add1bit(B[1] and A[nn-1], A[nn-1] and B[0], r[0,nn-1]);
-- remplace A[nn-1]and B[0] par 0 pour un multiplicande en base 2 (non signe)

x[0,nn]= r[0,nn] or ((x[0,nn-1])and(A[nn-1]));
-- remplace x[0,nn-1] and A[nn-1] par 0 pour un multiplicande en base 2 (non signe)

-- etages de 1 a nn-2-------
r[1..nn-2,0]= false^(nn-2);--attention parenthese obligatoire

x[1..nn-2,nn]=r[1..nn-2,nn] or (x[1..nn-2,nn-1]and (A[nn-1])^(nn-2)) ;
```

```
-- remplace x[1..nn-2,nn-1] and (A[nn-1])^(nn-2) par 0 pour un multiplicande en base 2 (non signe)

taband=tabandrec(nn, nn-2, A[0..nn-1],B[2..nn-1]);
(x[1..nn-2,0..nn-1],r[1..nn-2,1..nn])= add1bit(taband, x[0..nn-3,1..nn], r[1..nn-2,0..nn-1]);

--les sorties
S[0]= A[0] and B[0];
S[1..nn-1]= x[0..nn-2,0];
S[nn..(2*nn-1)]= x[nn-2,1..nn];
tel;

-- instantiation du multnbits generique
const a=3;
node instmultnbits(op1, op2: bool^a)
returns(res: bool^(2*a));
let
res= multnbits(a, op1, op2);
tel;
```