heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

ReDS

Systèmes numériques

Chapitre X

Numération et arithmétique

Serge BOADA
1993

Révision février 2011

Hes·so
Haute Ecole Spécialisée
de Suisse occidentale

Le présent manuel est une version informatique du manuel écrit par Serge Boada en 1993. Je remercie son auteur pour le travail réalisé.

Mise à jour de ce manuel,

Je remercie tous les utilisateurs de ce manuel de m'indiquer les erreurs qu'il comporte. De même, si des informations semblent manquer ou sont incomplètes, elles peuvent m'être transmises, cela permettra une mise à jour régulière de ce manuel.

Dernière mise à jour le 17 février 2011

Etienne Messerli

Contact: Etienne Messerli
e-mail : etienne.messerli@heig-vd.ch
Tél.: +41 (0)24 / 55 76 302

Coordonnées à la HEIG-VD :

Institut REDS
HEIG-VD,
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud
Route de Cheseaux 1
CH-1400 Yverdon-les-Bains
Tél : +41 (0)24 / 55 76 330
Internet : http://www.heig-vd.ch

Institut REDS
Internet : http://www.reds.ch

Autres personnes à contacter:

M. Yoan Graf yoan.graf@heig-vd.ch Tél direct +41 (0)24/55 76 259
M. Cédric Bardet cedric.bardet@heig-vd.ch Tél direct +41 (0)24/55 76 251

TABLE DES MATIERES

X.1 INTRODUCTION 4

X.2 NUMERATION DE POSITION....4

X.3 CHANGEMENTS DE BASE 4

X.4 ADDITION BINAIRE 8

X.5 RETENUE ANTICIPEE 9

X.6 SOUSTRACTION ET COMPLEMENT A 2 11

X.7 AUTRES REPRESENTATIONS DES NOMBRES NEGATIFS 12

X.8 DEPASSEMENT DE CAPACITE....12

X.9 MULTIPLICATION D'ENTIERS BINAIRES SANS SIGNE 13

X.10 MULTIPLICATION D'ENTIERS EN COMPLEMENT A 2 16

X.11 DIVISION D'ENTIERS BINAIRES SANS SIGNE....18

X.12 DIVISION D'ENTIERS EN COMPLEMENT A 2....23

X.13 STRUCTURE D'UNE RALU....23

X.14 VIRGULE FIXE ET VIRGULE FLOTTANTE 29

X.15 FORMATS STANDARD EN VIRGULE FLOTTANTE....30

X.16 LES OPERATIONS SUR LES NOMBRES EN VIRGULE FLOTTANTE 33

X.17 BITS DE GARDE ET ARRONDIS 38

X.18 EXERCICES....39

X.1 INTRODUCTION

Dans le chapitre sur les machines séquentielles complexes, nous avons été amenés à réaliser des opérations d'addition et de soustraction en binaire. Mais nos connaissances actuelles ne nous permettent guère de faire des calculs plus complexes, ne serait-ce qu'une multiplication ou une division, de manière efficace. Pas plus qu'elles ne nous permettent de comprendre la structure d'une RALU, qui est justement très influencée par les besoins liés à la multiplication et à la division. Avant de nous attaquer à la conception d'un processeur, il est maintenant temps de combler cette lacune.

Jusqu'ici, nous avons utilisé des mots binaires essentiellement dans le but de coder l'information. Nous allons étudier maintenant leur utilisation pour la représentation des nombres.

X.2 NUMÉRATION DE POSITION

Un système de numération est un langage consistant en une liste ordonnée de symboles appelés chiffres (digits), des règles définissant leur utilisation pour créer des nombres, ainsi que des règles définissant un jeu d'opérations telles que l'addition, la multiplication, etc., appelé arithmétique.

Le nombre de symboles différents est appelé base. Ainsi, en base 10 (ou décimal), nous avons dix symboles ou chiffres, alors qu'en base 2 (ou binaire), nous n'en avons que deux.

Le nombre de symboles différents étant très inférieur aux valeurs quantitatives que l'on désire représenter, on utilise la juxtaposition de chiffres pour créer des nombres. En général, chaque chiffre dans un nombre se voit attribuer un facteur de pondération, ou "poids", qui dépend de sa position dans le nombre. C'est ce que nous appellerons des systèmes de numération de position. Le binaire pur, le décimal ou l'hexadécimal font partie de cette catégorie, mais pas le code de Gray.

A l'exception de quelques codes BCD spéciaux qui n'ont plus guère d'intérêt aujourd'hui, comme l'excédent-de-trois ou le 4-2-2-1, les systèmes de numération de position utilisent une pondération exprimée en puissances croissantes de la base, la plus faible à droite et la plus forte à gauche. Ainsi, en base dix, le nombre 1993 correspond à l'expression polynomiale $1 \cdot 10^3 + 9 \cdot 10^2 + 9 \cdot 10^1 + 3 \cdot 10^0$.

Dans ce qui suit, nous nous restreindrons aux bases 2, 10 et 16, de loin les plus utilisées en électronique numérique.

X.3 CHANGEMENT DE BASE

Du fait que notre base de travail habituelle est la base 10 et que les machines de traitement de l'information travaillent en base 2, il est souvent nécessaire de procéder à des changements de base.

Bien évidemment, une conversion par substitution à l'aide d'une table (ce dont nous nous étions contentés jusqu'ici) n'est pas utilisable pour de grands nombres. Il nous faudra donc des méthodes de conversion algorithmiques. Celles-ci font intervenir des calculs

arithmétiques qui seront effectués en binaire par une machine électronique, mais en décimal si nous travaillons "à la main".

a) conversion "à la main" de base 10 en base 2

D'une façon générale, un nombre entier en base b, que nous désignerons par E_b, peut s'écrire sous la forme polynomiale :

$$E_b = d_m \cdot b^m + d_{m-1} \cdot b^{m-1} + ... + d_1 \cdot b^1 + d_0 \cdot b^0$$

où d_m à d_0 sont les différents chiffres en base b constituant le mot qui représente le nombre.

Mais nous pouvons également écrire :

$$E_b = ((...((d_m \cdot b + d_{m-1}) \cdot b + d_{m-2}) \cdot b + ... + d_1) \cdot b + d_0 \ .$$

Des divisions successives par la base b nous donneront des restes successifs d_0, puis d_1, … d_m. Nous pouvons donc procéder de cette manière pour passer "à la main" de base 10 en base 2, en effectuant bien sûr les calculs en base 10.

Calculons par exemple le nombre 1993 en base 2 :

Nous obtenons : $1993_{10} = 111\ 1100\ 1001_2$

÷ 2	restes
1993	1
996	0
498	0
249	1
124	0
62	0
31	1
15	1
7	1
3	1
1	1

La partie fractionnaire d'un nombre peut s'écrire :

$$F_b = d_{-1} \cdot b^{-1} + d_{-2} \cdot b^{-2} + ... + d_{-k} \cdot b^{-k}$$

mais également :

$$Fb = ((...((d_{-k} \cdot b^{-1} + d_{-k+1}) \cdot b^{-1} + d_{-k+2}) \cdot b^{-1} + ... + d_{-1}) \cdot b^{-1}$$

En multipliant la partie fractionnaire par la base, nous obtenons d^{-1} comme partie entière. Une nouvelle multiplication par b de la partie fractionnaire résultante donnerait d_{-2} comme partie entière, et ainsi de suite. Par multiplications successives nous obtiendrons donc les chiffres de la partie fractionnaire d'un nombre.

Calculons par exemple la fraction 0.8

	partie fractionnaire	partie entière	
x 2	0,8		
	0,6	1	→ d_{-1}
	0,2	1	→ d_{-2}
	0,4	0	→ d_{-3}
	0,8	0	→ d_{-4}
	⋮	⋮	

Le calcul ne se termine pas. Nous obtenons :

$$0.8_{10} = 0.1100 \quad 1100 \quad 1100_2 \quad \ldots = 0.\overline{1100}_2$$

b) Conversion "à la machine" de base 10 en base 2

Dans la machine, les nombres décimaux seront bien sûr représentés en binaire, soit en BCD, soit en ASCII. La conversion ASCII → BCD n'offrant aucune difficulté, nous nous concentrons sur la conversion BCD → binaire pur.

Les expressions pour la partie entière et pour la partie fractionnaire d'un nombre, obtenues par mises en évidence successives de la base b ou de son inverse b^{-1}, que nous avons utilisé sous a), nous donnent directement des algorithmes de conversion.

Pour la partie entière, avec un nombre de m+1 chiffres BCD, nous pouvons utiliser l'algorithme ci-dessous, écrit en pseudo Ada :

```
resultat := 0;
for i in m downto 0 loop
   resultat := resultat * 1010₂ + chiffre_BCD_de_rang [ i ];
   -- le calcul est effectue en binaire
end loop;
```

Pour convertir 1993_{10} en binaire, nous aurions ainsi la suite de calculs :

$$((\underbrace{0001}_{\text{BCD rang 3}} \times 1010 + \underbrace{1001}_{\text{BCD rang 2}}) \times 1010 + \underbrace{1001}_{\text{BCD rang 1}}) \times 1010 + \underbrace{0011}_{\text{BCD rang 0}}$$

Pour la partie fractionnaire, nous pouvons utiliser l'algorithme suivant (en pseudo Ada) :

```
resultat := 0;
-- k chiffres decimaux apres la virgule
for i in k downto 1 loop
   resultat := (resultat  + chiffre_BCD_de_rang[ -i ] ) / 1010₂ ;
   -- le calcul est effectue en binaire
end loop;
```

Cependant, ces divisions par 1010_2 seront souvent sans fin, ce qui provoquera une accumulation d'erreurs d'arrondi ou de troncature. Il peut être plus judicieux de procéder comme pour la partie entière, puis de diviser une seule fois à la fin par dix à la puissance k (en binaire bien sûr, et cela n'est pas un simple décalage de la virgule).

La méthode ci-dessus est la plus utilisée dans les systèmes à microprocesseur. Elle pose cependant le problème de la multiplication et de la division en binaire, problème dont nous nous occuperons plus loin. Dans l'exercice X.18.1, nous verrons une autre méthode algorithmique sans multiplication ni division.

L'implémentation de ces algorithmes fait appel à des machines séquentielles complexes, alors que le problème de la conversion BCD → binaire est de type combinatoire. C'est là un exemple de décomposition en séquence d'un problème combinatoire, dans le but de diminuer le coût du circuit.

c) Conversion " à la main " de base 2 en base 10

Pour convertir un nombre binaire en décimal, il suffit d'additionner les poids respectifs des " 1 " apparaissant dans la représentation binaire. Une table des puissances de deux nous facilite donc la tâche.

Nous aurons, par exemple :

$1\ 1\ 1\ 1\ 1\ 0\ 0\ 1\ 0\ 0\ 1_2$

x 2^0	1
x 2^1	0
x 2^2	0
x 2^3	8
x 2^4	0
x 2^5	0
x 2^6	64
x 2^7	128
x 2^8	256
x 2^9	512
x 2^{10}	1'024
total :	1'993

d) Conversion " à la machine " de base 2 en base 10

En fait, on réalise une conversion de binaire en BCD naturel, puisqu'une machine binaire ne peut pas générer directement des chiffres décimaux.

Quelle que soit la base dans laquelle on désire exprimer un nombre, il est possible d'effectuer la conversion selon des algorithmes similaires à ceux que nous avons décrits pour la conversion de base 10 en base 2 (cas b)).

Mais les calculs doivent être effectués dans la base de destination, donc en BCD pour une conversion de base 2 en base 10. Ces calculs sont évidemment différents de ceux effectués sur des nombres binaires purs. Ils demandent des circuits ou des algorithmes différents.

En s'inspirant de la méthode utilisée pour passer "à la main" de base 10 en base 2 (cas a)), nous pouvons effectuer les calculs en base 2. C'est par une suite de divisions par 1010 (dix) que nous obtiendrons des restes correspondants aux chiffres successifs de la partie entière d'un nombre, le chiffre de poids faible d'abord. Par une suite de multiplications par 1010 de la partie fractionnaire en base 2, nous obtiendrons des parties entières successives correspondant aux chiffres de la partie fractionnaire en base 10, le poids fort d'abord.

Dans l'exercice X.18.3, nous verrons une autre méthode algorithmique, qui ne nécessite ni multiplication ni division.

Ces conversions peuvent aussi s'effectuer de façon purement combinatoire à l'aide de circuits standard spécifiquement prévus à cet effet, tels que les 74184 et 74484 pour le passage de BCD à binaire, et les 74185 et 74485 pour le passage de binaire à BCD. Mais il faut déjà 5 circuits à 20 pattes de type 74484 pour convertir en binaire un nombre de 4 décades, et 12 circuits pour un nombre de 6 décades ! Comme ces conversions interviennent généralement dans une interface homme-machine, elles n'ont pas besoin d'être effectuées très vite et peuvent donc s'accommoder d'un traitement séquentiel, plus économique.

X.4 ADDITION BINAIRE

Pour additionner deux nombres décimaux à la main, nous procédons par additions successives des chiffres de même poids (en commençant par le poids faible) et du report de l'addition précédente. La même méthode peut être appliquée à l'addition en binaire. Par exemple :

	1 1		1 1 1 1	1	1	← reports
1er nombre :	1986_{10}	=	0 1 1 1	1 1 0 0	0010_2	
2e nombre :	90_{10}	=	0 0 0 0	0 1 0 1	1010_2	
somme :	2076_{10}	=	1 0 0 0	0 0 0 1	1100_2	

Il nous suffit de disposer d'un circuit permettant d'additionner deux bits d'entrée et un report, puis de cascader autant de circuits qu'il y a de bits dans le plus grand nombre envisagé. Cet additionneur à deux bits et un report répond à la table X.1 ci-dessous :

A_N	B_N	C_N	S_N	C_{N+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Fig. X.1

Pour additionner deux nombres A et B de quatre bits, nous pouvons donc utiliser un circuit selon le schéma bloc de la figure X.2. Il faut cependant constater qu'avec A = 0111 et B = 0001, par exemple, le report devra se propager à travers les quatre cellules.

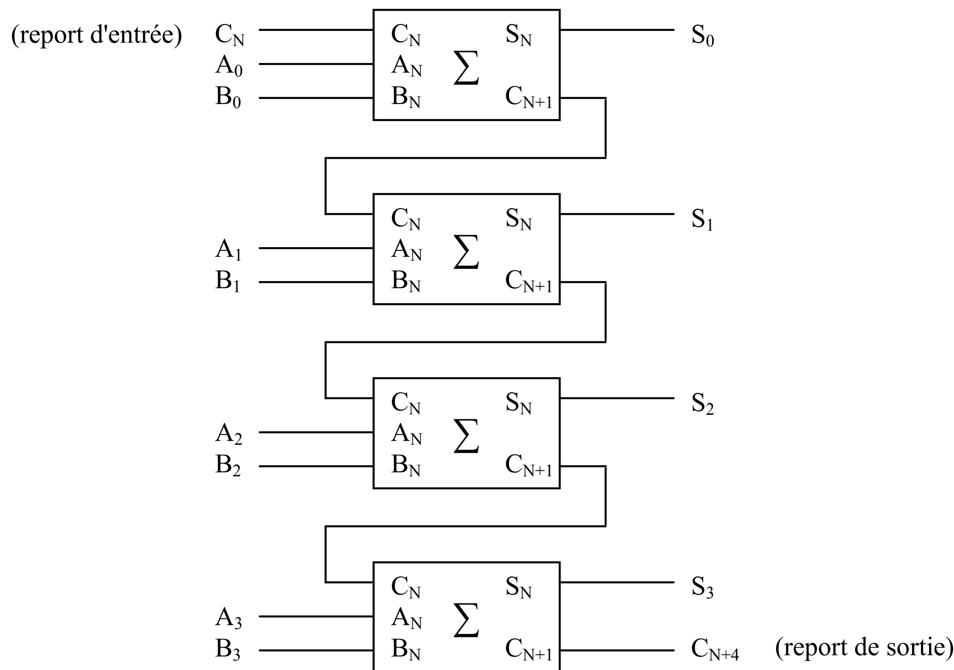

Fig. X.2

Cette configuration est donc économique mais donne lieu à un temps de réaction assez long. Un premier pas vers une accélération du fonctionnement consiste à utiliser un additionner de deux nombres de quatre bits, tel le 74LS283 que nous connaissons déjà, conçu globalement à partir d'une table de vérité à 9 variables d'entrée et non pas comme une cascade d'additionneurs à un bit. S'il s'agit d'additionner des nombres de plus de quatre bits, en cascadant des 74LS283 nous aurons cependant encore un assez long temps de propagation du report.

X. 5 RETENUE ANTICIPÉE

La technique d'anticipation du report (carry look-ahead) nous permet d'accélérer la propagation du report d'un additionneur intégré au suivant. Pour comprendre cette technique, examinons de plus près la table X.1 : les deux premières lignes ne génèrent pas de report, les quatre suivantes génèrent un report s'il y avait un report d'entrée, nous dirons qu'il y a propagation du report, alors que les deux dernières lignes génèrent de toutes façons un report.

Appelons G_N la fonction de génération du report et P_N la propagation.

Nous avons $G_N = A_N \text{ and } B_N$ et $P_N = A_N \text{ xor } B_N$

Ces deux fonctions ne dépendent donc pas du report d'entrée.

Nous aurons un report $C_{N+1} = G_N$ or (P_N and C_N).

Pour un additionneur de quatre bits, tel celui contenu dans l'ALU 74F381 (similaire à la 74F382 que nous avons vue au chapitre VII, mais générant G et P à la place du report C_{N+4} et du dépassement de capacité OVF, voir figure X.3 a), ce raisonnement reste valable. En mettant en parallèle trois 74F381 et un 74F382 comme nous le montre la figure X.4, de façon à pouvoir effectuer des opérations combinatoires sur des nombres de 16 bits, nous pouvons utiliser un circuit de calcul de la retenue comme le 74F182 (voire figure X.3 b). Ce circuit effectue les calculs suivants :

$C_{N+X} = G_O$ or (P_O and C_N) = report d'entrée dans le 2ème F381

$C_{N+Y} = G_1$ or (P_1 and C_{N+X}) = G_1 or (P_1 and G_O) or (P_O and P_1 and C_N) = report d'entrée dans le 3ème F381

$C_{N+Z} = G_2$ or (P_2 and C_{N+Y}) = G_2 or (P_2 and G_1) or (P_2 and P_1 and G_0) or (P_2 and P_1 and P_0 and C_N) = report d'entrée dans le F382.

Les différents G_i et P_i étant générés par des sommes de produits, nous constatons que le calcul d'un report ne nécessite que le temps de passage à travers quatre portes (deux pour les G_i et P_i, deux de plus pour les C_i) quel que soit son rang. A partir de trois additionneurs en parallèle, une unité d'anticipation des reports comme la F182 permet donc d'accélérer le calcul.

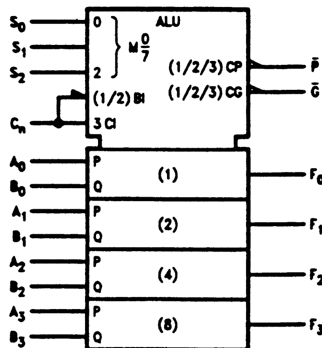

Fig. X.3 a

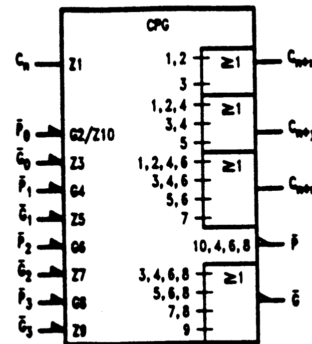

Fig. X.3 b

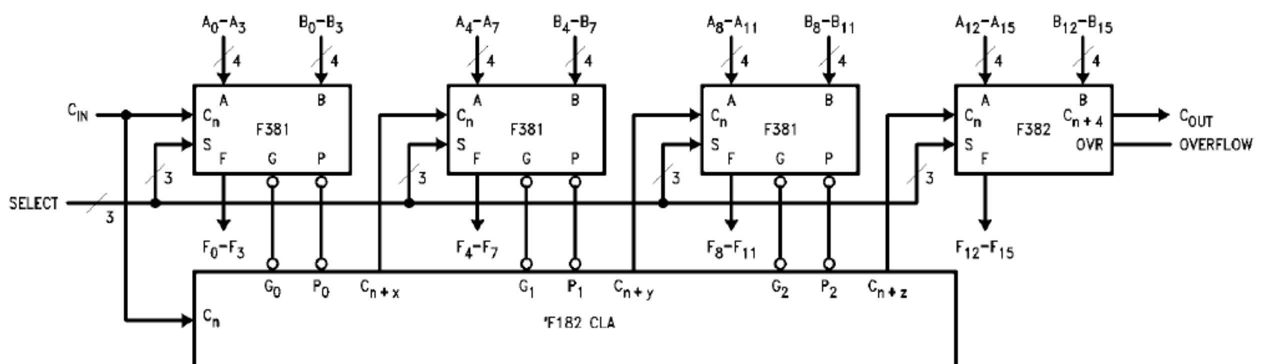

FIGURE 1. 16-Bit Lookahead Carry ALU Expansion

Fig. X.4

X.6 SOUSTRACTION ET COMPLÉMENT À 2

Une méthode similaire à l'addition peut être utilisée pour la soustraction. C'est cependant une méthode différente que nous allons appliquer car elle nous évitera d'avoir à développer des circuits spécifiques pour la soustraction. Pour cela, il nous faut d'abord parler de la représentation des nombres négatifs.

Prenons un compteur-décompteur de 4 bits, binaire, et appliquons une impulsion de décomptage à partir de l'état 0000. Nous savons bien qu'il va se boucler sur la valeur 1111 que nous pouvons donc assimiler à la valeur négative –1. Une deuxième impulsion de décomptage donne 1110 que nous assimilerons à –2, etc. … . Nous pouvons convenir de considérer comme négatives les valeurs 1111, 1110, … jusqu'à 1000 et positives les valeurs de 0000 à 0111. Ainsi, les valeurs négatives ont le bit de poids le plus fort à 1 et les valeurs positives ont ce même bit à 0.

Additionnons maintenant +1 et –1, nous obtenons :

```
    0 0 0 1
  + 1 1 1 1
  ---------
[1] 0 0 0 0
```

Dans une représentation de nos nombres à l'aide de 4 bits seulement, le résultat de cette addition est bien 0000, le "1" de poids 2^4 étant en fait le report de sortie. En travaillant en modulo 2^n , ou n est le nombre de bits utilisé pour représenter nos nombres, et en utilisant la représentation des nombres négatifs expliquée ci-dessus, nous pouvons bien remplacer la soustraction d'un nombre par l'addition du nombre complémentaire. Bien sûr, cela n'a d'intérêt que si ce nombre complémentaire peut s'obtenir facilement. C'est bien le cas : nous pouvons facilement vérifier qu'il suffit d'inverser tous les bits puis d'ajouter 1. Par exemple, dans une représentation à 8 bits, -25 s'obtient en faisant :

```
            25 =   0 0 0 1  1 0 0 1
     inversion :   1 1 1 0  0 1 1 0
 additionner 1 :   1 1 1 0  0 1 1 1  =  -25

        preuve :   0 0 0 1  1 0 0 1
                 + 1 1 1 0  0 1 1 1
                 ------------------
                [1] 0 0 0 0  0 0 0 0
```

Cette représentation des nombres négatifs binaires est appelée "complément à 2". Mais on devrait en fait parler de complément à 2^n, puisque l'addition d'un nombre et de son complément donne 2^n, ce qui correspond à zéro en modulo 2^n.

N.B. Il ne faut pas confondre "un nombre X noté en complément à 2" et "le complément à 2 d'un nombre X". Par exemple, le nombre 1110 0111 en notation complément à 2 = -25, alors que le complément à 2 de 1110 0111 = à 0001 1001, soit +25.

Pour en revenir à notre soustraction, nous la réaliserons à l'aide d'un additionneur, en ajoutant le complément à 2 du nombre à soustraire. Comme le complément à 2 s'obtient en inversant tous les bits puis en ajoutant 1, il nous suffira d'inverser effectivement tous les bits du nombre à soustraire, et d'effectuer l'addition en forçant à 1 le report d'entrée de l'additionneur (celui de poids faible, s'il est réalisé avec plusieurs circuits en parallèle comme dans la figure X.4).

X.7 AUTRES REPRÉSENTATIONS DES NOMBRES NÉGATIFS

La représentation dite "en complément à 2" est la plus utilisée. Mais nous trouverons aussi des applications à la représentation dite "en complément à 1". Il s'agit en fait du complément à 2^n-1, plus facile à calculer puisqu'il suffit d'inverser tous les bits. Cette représentation comporte cependant deux inconvénients : elle ne correspond pas au fonctionnement des compteurs-décompteurs standard, et elle donne lieu à une double représentation du zéro. Ainsi, en complément à 1, le complément de 0 (soit –0) noté sur 8 bits sera 1111 1111 : + 0 et – 0 ne s'écrivent pas de la même façon !

Pour la mantisse des nombres réels en notation scientifique (voir plus loin), en particulier, nous utiliserons aussi une représentation dite "en signe et amplitude" : le bit le plus à gauche (à la place habituelle du bit de poids fort) indique le signe, alors que les bits suivants indiquent l'amplitude ou valeur absolue. Comme pour les notations en complément à 2 et en complément à 1, le bit "de poids fort" est à 1 pour un nombre négatif, et à 0 pour un nombre positif.

Les exposants des nombres réels en notation scientifique sont généralement exprimés dans une notation biaisée, plus connue sous le nom de "excédent de x", où x est généralement égal à 2^{n-1}-1, n étant le nombre de bits utilisés. Par exemple, pour une représentation sur 8 bits, on parlera d'excédent de 127 : la représentation est obtenue en ajoutant 127 au nombre désiré.

Le principal intérêt de cette notation biaisée est que les comparaisons peuvent être effectuées avec des comparateurs prévus pour des nombres sans signe, comme le 74LS85, ce qui n'est pas le cas des autres notations ci-dessus.

X.8 DÉPASSEMENT DE CAPACITÉ

Dans la notation en complément à 2, avec 8 bits à disposition, par exemple, nous ne pouvons représenter des entiers signés qu'entre +127 et –128. En effectuant le calcul +120 +9, nous dépasserons donc la capacité de représentation. En effet :

```
 +120  =  0 1 1 1  1 0 0 0
   +9  =  0 0 0 0  1 0 0 1
 total    1 0 0 0  0 0 0 1   =  -127
```

De même, le calcul –120 –9 donnera un dépassement de capacité ou débordement (overflow) :

```
 -120  =    1 0 0 0  1 0 0 0
   -9  =    1 1 1 1  0 1 1 1
 total  [1] 0 1 1 1  1 1 1 1   =  +127
         ↑_ report
```

Le débordement se traduit par une erreur de signe du résultat. Contrairement au cas des nombres entiers sans signe, le débordement dans les nombres entiers signés en complément à 2 ne peut pas être détecté à l'aide du report de sortie uniquement. Nous venons en effet de voir deux cas de débordement, l'un générant un report et l'autre pas.

Pour établir l'expression du débordement, il faut examiner les 3 cas qui peuvent se présenter lors de l'addition :

a) si l'on additionne un nombre positif et un nombre négatif, il ne peut pas y avoir de débordement; puisque les bits de poids fort sont l'un à 0 et l'autre à 1, un report de poids 2^{n-1} donnera également un report de poids 2^n.

b) si les deux nombres sont positifs, il ne peut pas y avoir de report de poids 2^n puisque les bits de poids fort sont tous deux à 0; s'il y a un report de poids 2^{n-1}, le résultat apparaît comme négatif puisque le bit de poids fort sera à 1, et il y a bien un dépassement de capacité.

c) si les deux nombres sont négatifs, il y aura toujours un report de poids 2^n puisque les deux bits de poids fort sont à 1; le bit de signe du résultat sera à 0 (positif) s'il n'y a pas de report de poids 2^{n-1}, ce qui constitue un dépassement de capacité.

Nous constatons qu'une addition de deux nombres de n bits donne un dépassement de capacité lorsque le bit de report de poids 2^n est différent de celui de poids 2^{n-1}. D'où l'expression : $OVF = C_n \text{ xor } C_{n-1}$.

La plupart des circuits arithmétiques ont une sortie indiquant un éventuel dépassement de capacité, comme nous l'avons vu, par exemple, dans le 74F382.

X.9 MULTIPLICATION D'ENTIERS BINAIRES SANS SIGNE

L'opération de multiplication, de par sa complexité, est presque toujours décomposée en opérations plus simples (sauf pour de très petits nombres). Cette décomposition peut être combinatoire (dite aussi "spatiale") ou séquentielle (dite aussi "temporelle"). Pour en comprendre les idées de base, il est bon de se remémorer la méthode de multiplication "à la main" que nous avons apprise à l'école primaire.

Analysons par exemple la multiplication 125 x 139

```
        1 2 5
      x 1 3 9
      -------
      1 1 2 5  ←  9 x 5 + 9 x 2 x 10 + 9 x 1 x 100
  +     3 7 5 0  ←  (3 x 5 + 3 x 2 x 10 + 3 x 1 x 100) x 10
  +   1 2 5 0 0  ←  (1 x 5 + 1 x 2 x 10 + 1 x 1 x 100) x 100
      ---------
      1 7 3 7 5
```

En fait, nous nous contentons d'effectuer des multiplications chiffre par chiffre (à l'aide de tables de multiplication) et des additions, après avoir tenu compte des poids respectifs des chiffres en multipliant par la puissance de 10 correspondante (ce qui se fait en décalant simplement le nombre sur la gauche).

En binaire, la table des multiplications chiffre par chiffre se réduit à 0 x 1 = 0 et 1 x 1 = 1. Il n'y a pas de report possible sur le chiffre de rang supérieur comme c'est le cas en décimal (9 fois 5 égale 45, on écrit 5 et on reporte 4).

La multiplication par la base, soit 2, se résume à un décalage à gauche, le nouveau bit de poids faible ainsi obtenu valant 0, tout comme en décimal. C'est le cas, du moins, si nous

nous limitons aux entiers sans signe. Nous traiterons plus loin le cas des nombres entiers signés, en complément à 2.

Les diverses additions seront bien sûr effectuées par des additionneurs.

En se basant sur la façon de multiplier à la main, nous pouvons donc établir une structure modulaire purement combinatoire composée de multiplicateurs par 1 et d'additionneurs. Réalisée avec des circuits MSI et LSI, une telle structure prendrait rapidement des proportions gigantesques, puisque le nombre de modules augmente avec le carré du nombre de bits des opérandes. Par contre, la régularité de cette structure se prête bien à une intégration à large échelle. Il est ainsi possible d'obtenir un circuit de multiplication 32 bits par 32 bits en un seul boîtier, comme le Am29323, qui permet d'effectuer une multiplication en 80 nanosecondes.

Si nous ne visons pas une performance maximale, nous pouvons réaliser une multiplication de façon plus économique en procédant à un décomposition en séquence. Cette multiplication sera réalisée par une machine séquentielle dotée d'une unité de traitement universelle, et non pas par un circuit spécifique.

Notre façon de multiplier à la main peut être traduite dans l'algorithme ci-dessous, valable pour des entiers sans signe (ou positifs) :

```
resultat:= 0;
while multiplicateur /= 0 loop
   multiplicateur:= multiplicateur/2;
   (* reste = bit de poids faible*)
   if reste=1 then résultat:= résultat + multiplicande;
   end if;
   multiplicande:= multiplicande*2;
end loop;
```

Au lieu d'effectuer les diverses additions avec autant d'additionneurs, comme dans la décomposition combinatoire, il suffit d'additionner successivement les valeurs voulues à l'aide d'un seul additionneur. Trois registres suffisent, pour contenir le multiplicande, le multiplicateur et le résultat. Celui contenant le résultat est un accumulateur, tel que nous l'avons vu au § VII.27, puisqu'il fournit un des opérandes de l'addition et en mémorise le résultat. Les deux autres doivent pouvoir effectuer un décalage série.

Nous établirons le schéma bloc d'une MSS implémentant cet algorithme, en exercice.

Dans le cas général, la multiplication de 2 nombres de n bits donne un résultat comportant $2 \cdot n$ bits. Si l'on applique l'algorithme ci-dessus à la lettre, il faut donc disposer de deux registres de $2 \cdot n$ bits, pour le résultat et le multiplicande décalé, et d'un registre de n bits pour le multiplicateur. De plus, l'additionneur doit permettre l'addition de 2 nombres de $2 \cdot n$ bits chacun. Nous pouvons réduire nos besoins à 3 registres de n bits, dont deux connectés en série comme indiqué à la figure X.5, et un additionneur de n bits de large seulement, en raffinant notre algorithme à l'aide des observations suivantes :

a) le nombre de bits du multiplicateur diminue à chaque tour dans la boucle **while**, puisque le multiplicateur est chaque fois décalé à droite (divisé par 2);

b) le nombre de bits du résultat augmente de un, au pire, à chaque tour dans la boucle;

c) à chaque tour, seuls n bits du résultat partiel participent à l'addition, s'il y a lieu, et un nouveau bit (à partir du poids le plus faible) atteint sa valeur définitive.

Il est donc possible de décaler à chaque tour le résultat partiel vers la droite dans le registre contenant le multiplicateur, ce qui permet de conserver les bits du résultat qui ne seront plus modifiés. Du même coup, ce décalage aligne les n bits du résultat partiel qui doivent participer à l'addition avec le multiplicande.

L'algorithme ainsi obtenu est le suivant :

```
resultat_haut:= o
for i in 1 to n loop     -- n = nb. de bits
   if bit_de_poids_faible(multiplicateur) = 1
   then resultat_haut:= resultat_haut + multiplicande;
-- le report est memorise dans un flip-flop
   else resultat_haut:= resultat_haut + 0;
   decale_a_droite(report,resultat_haut,multiplicateur);
-- le report precedent va dans le Msbit de resultat_haut
end;
```

La figure X.5 nous montre un schéma bloc très simplifié de ce que pourrait être l'UT d'une MSS effectuant cette multiplication.

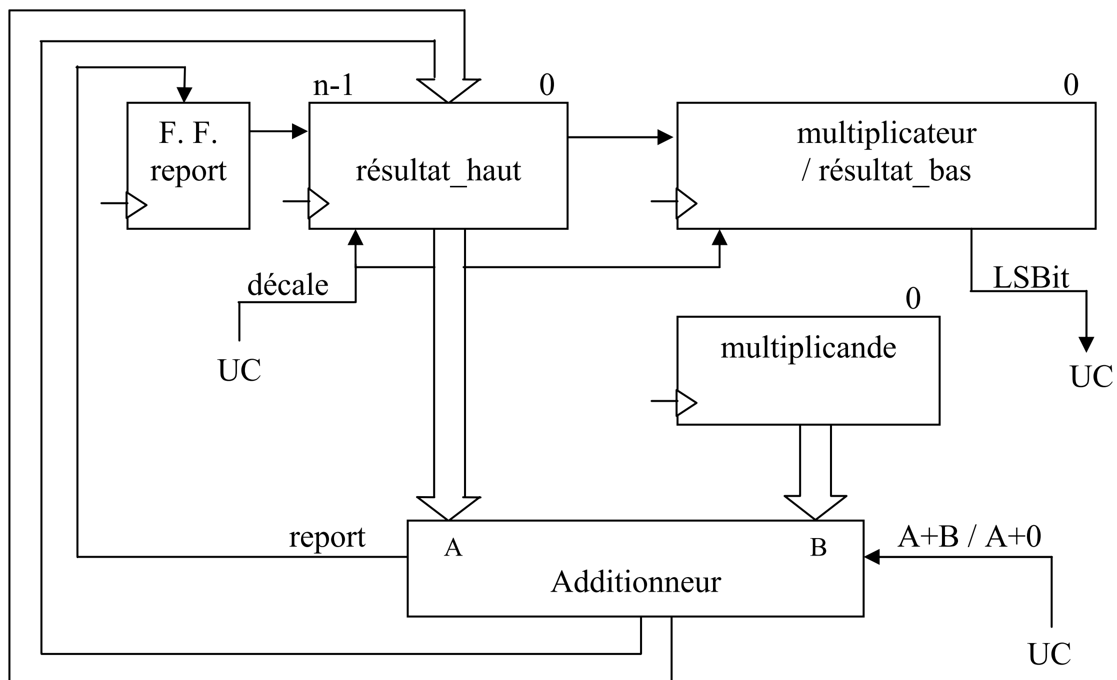

Fig. X.5

X.10 MULTIPLICATION D'ENTIERS EN COMPLÉMENT À 2

L'addition et la soustraction s'effectuent de la même façon (avec les mêmes circuits) que ce soit sur des nombres sans signe ou des nombres en complément à 2. Il n'y a que le dépassement de capacité qui diffère. Malheureusement, il n'en va pas de même pour la multiplication (ou la division). Nous pouvons facilement nous en convaincre à l'aide d'un exemple :

Multiplions 1011 (onze) par 1101 (treize).

```
          1 0 1 1
        x 1 1 0 1          1 1        -5
        ---------        x 1 3      x -3
          1 0 1 1        -----      ----
      1 0  1 1             3 3        1 5
    1 0 1  1              1 1 -
  ---------------        -----
  1 0 0 0  1 1 1 1   =   1 4 3
```

Considérés comme des nombres sans signe, 1011 x 1101 nous donnent 1000 1111, ce qui correspond bien à 11 x 13 = 143. Mais considérés comme des nombres en complément à 2, 1011 représente –5 et 1101 représente –3. Leur produit devrait pouvoir se lire + 15, alors que nous obtenons – 113 !

Si nous voulons utiliser le même algorithme que pour les nombres sans signe, il faudra prendre la valeur absolue des opérandes (donc prendre le complément à 2 des opérandes négatifs), effectuer la multiplication des valeurs absolues comme nous l'avons fait au paragraphe X.9, calculer le signe du résultat final et, s'il est négatif, prendre le complément à 2 du résultat de la multiplication des valeurs absolues. Les éventuelles complémentations et le calcul du signe vont bien sûr rallonger le temps de calcul.

Il existe divers algorithmes, plus efficaces que la méthode ci-dessus, permettant d'éviter les complémentations. Nous en étudierons un, celui de Robertson, très utilisé. Pour bien le comprendre, refaisons le calcul 1011 x 1101 en considérant qu'il s'agit de nombres exprimés dans la notation en complément à 2.

```
          1 0 1 1  =   -5
        x 1 1 0 1  =   -3
        ---------
  1 1 1 1  1 0 1 1          résultats partiels négatifs exprimés sur 8 bits
+ 1 1 1 0  1 1
+ 1 1 0 1  1
  ---------------
  1 0 1 1  1 1 1 1
- 1 0 1 1  0 0 0 0          correction – (multiplicande x 2^n)
  ---------------
  0 0 0 0  1 1 1 1  =  + 15
```

Comme le multiplicande et le multiplicateur comportent 4 bits, le résultat en comportera 8. Nous exprimerons nos résultats partiels sur 8 bits, en tenant compte du signe : si le multiplicande est négatif, les bits à gauche des résultats partiels sont à 1 et non pas à 0. En effet 1111 1011 représente bien –5 sur 8 bits en complément à 2.

A la fin des additions des résultats partiels, nous constatons que notre résultat est incorrect. En fait, il faut encore lui soustraire 2^n fois le multiplicande. Cela s'explique facilement, si l'on prend en considération que le multiplicateur est négatif. En effet, nous avons calculé

$A \cdot (2^n - |B|)$ puisque notre multiplicateur B est exprimé en complément à 2. Ce que nous voulons c'est $A \cdot (-|B|) = A \cdot (2^n - |B|) - A \cdot 2^n$. Il nous faut donc bien retrancher 2^n fois le multiplicande si le multiplicateur est négatif.

Nous pouvons éviter cette correction du résultat en observant qui si le multiplicateur est négatif, son bit de poids fort est à 1. C'est le bit de poids 2^{n-1}, donc nous avons ajouté un dernier résultat partiel qui est 2^{n-1} fois le multiplicande, et tout de suite après nous avons apporté une correction en soustrayant 2^n fois le multiplicande. Il est plus logique d'effectuer normalement les additions pour les n-1 bits de poids faible du multiplicateur, puis de soustraire 2^{n-1} fois le multiplicande, si le bit de poids fort du multiplicateur est à 1.

Comme pour les nombres sans signe, nous allons établir un algorithme nous permettant de travailler avec 3 registres de n bits, dont 2 connectés en série, et un additionneur de n bits. Pour cela il nous faudra décaler le registre double contenant le résultat et le multiplicateur, tout en conservant le signe du résultat. Ce n'est pas le report qu'il faudra décaler dans le bit le plus significatif du résultat, mais un bit de signe, qui correspond au **ou exclusif** du dépassement de capacité et de l'ancienne valeur de ce bit : MSBit(résultat) := OVF **ou exclusif** MSBit(résultat). Cela se vérifie facilement en examinant les divers cas possibles.

Nous obtenons donc l'algorithme suivant :

```
resultat_haut := 0;
for i in 1 to n-1 loop
-- n = nombre de bits
  if bit_de_poids_faible(multiplicateur)=1 then
    resultat_haut:= resultat_haut + multiplicande;
  else resultat_haut:= resultat_haut + 0;
  end if;
  decale_a_droite(overflow xor msbit, resultat_haut,
                  multiplicateur);
end loop;
-- overflow est calcule et memorise dans un flip-flop
-- lors de chaque operation arithmetique
if bit_de_poids_faible(multiplicateur)=1 then
  resultat_haut:= resultat_haut - multiplicande;
else resultat_haut:= resultat_haut - 0;
end if;
decale_a_droite(overflow xor msbit, resultat_haut,
                multiplicateur);
```

Pour effectuer des multiplications signées, le schéma bloc de l'UT que nous avons vu à la figure X.5 doit être légèrement modifié. Nous aurons besoin d'un additionneur-soustracteur (ou d'une ALU) au lieu d'un simple additionneur. Cet additionneur-soustracteur ou cette ALU doit générer un signal de dépassement de capacité, OVF. Ce signal doit être mémorisé dans un flip-flop. Le **ou exclusif** de OVF et du MSBit du registre Résultat_haut doit être connecté à l'entrée de ce registre, pour le décalage à droite.

Nous obtenons ainsi le schéma bloc très simplifié de l'UT de la figure X.6.

Il existe bien sûr d'autres algorithmes de multiplication, chacun offrant divers avantages. Nous étudierons celui de Booth dans l'exercice X.18.6.

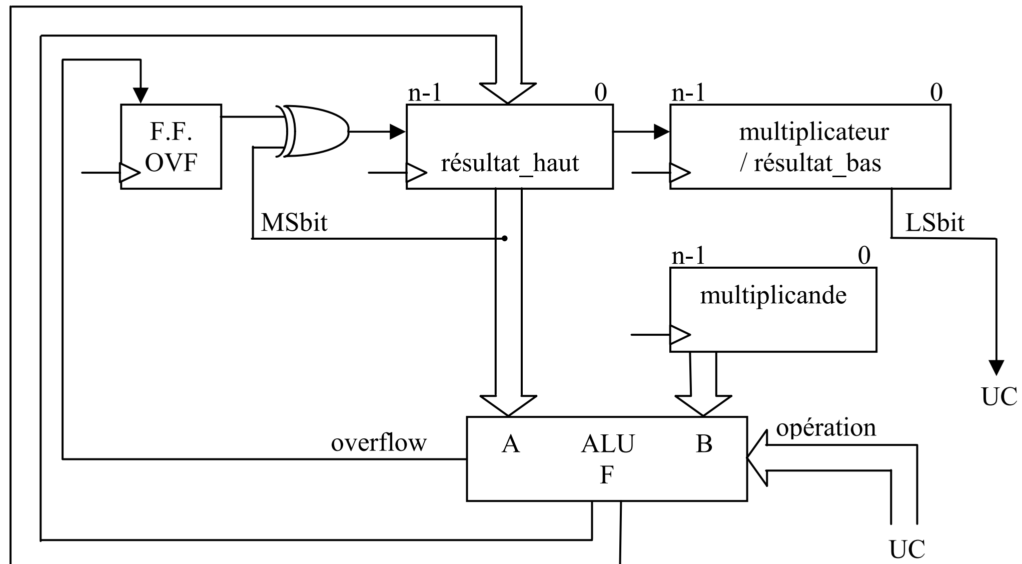

Fig. X.6

X.11 DIVISION D'ENTIERS BINAIRES SANS SIGNE

Comme pour la multiplication, nous allons établir un algorithme de division en étudiant notre façon de diviser à la main.

Prenons par exemple un nombre de 8 bits et divisons-le par un nombre de 4 bits, en visant un résultat exprimable sur 4 bits, et en détaillant toutes les opérations.

opération		***dividende*/restes***	********	***diviseur***	***quotient***
		1 0 1 1 1 0 0 1	*	1 1 0 1	
a	-	1 1 0 1 0 0 0 0			
b	1	1 1 1 0 1 0 0 1			
c		1 0 1 1 1 0 0 1	**		
d	-	1 1 0 1 0 0 0			
e	0	0 1 0 1 0 0 0 1	**		1
f	-	1 1 0 1 0 0			
g	0	0 0 0 1 1 1 0 1	**		1 1
h	-	1 1 0 1 0			
I	0	0 0 0 0 0 0 1 1	**		1 1 1
j	-	1 1 0 1			
k	1	1 1 1 1 0 1 1 0			1 1 1 0
l		0 0 0 0 0 0 1 1	**		

Les opérations effectuées sont les suivantes :

a) dividende – (diviseur · 2^4)
b) si le résultat de cette opération est positif, le quotient sera supérieur ou égal à 2^4 et ne pourra être exprimé sur 4 bits ; ce n'est pas le cas dans notre exemple, puisqu'il y a un emprunt de sortie (bit encadré).
c) on reprend (on dit plutôt "restaure", "restore" en anglais) le dividende puisque la soustraction tentée n'était pas possible
d) on essaie dividende – (diviseur · 2^3)
e) le reste est positif (l'emprunt de sortie est nul), donc la soustraction est possible, donc le bit de poids 2^3 du quotient vaut 1, et le reste est conservé
f) on essaie reste – (diviseur · 2^2)
g) idem e), le bit de poids 2^2 du quotient est à 1
h) on essaie reste – (diviseur · 2^1)
i) idem e), le bit de poids 2^1 du quotient est à 1
j) on essaie reste – (diviseur · 2^0)
k) le reste est négatif (l'emprunt de sortie est à 1), donc la soustraction est impossible, donc le bit de poids 2^0 du quotient vaut 0
l) le reste obtenu sous i) est restauré; c'est le reste final.

Mis à part le test initial de dépassement de capacité que nous n'avons pas l'habitude de faire en décimal, le reste des opérations est conforme à notre façon de procéder lors d'une division à la main en base dix. Si la présentation diffère quelque peu, c'est que nous avons détaillé certaines étapes que nous effectuons d'ordinaire mentalement, en base dix. Il faut aussi remarquer qu'en binaire il suffit d'essayer de soustraire le diviseur fois 2^i pour trouver le bit de rang i du quotient, alors qu'en décimal nous devons essayer successivement de soustraire le diviseur neuf fois 10^i, puis, si le résultat est négatif, le diviseur fois huit fois 10^i, et ainsi de suite jusqu'au diviseur fois un fois 10^i.

Nous tirerons les enseignements suivants de cet exemple :

1) il est possible d'obtenir une division par une suite (ou une cascade) de soustractions et de décalages
2) les emprunts de sortie des soustractions sont égaux à l'inverse des bits correspondants du quotient
3) il n'est pas nécessaire d'effectuer les soustractions sur 8 bits (2 · n) mais seulement sur 5 bits n + 1); la soustraction du test de dépassement de capacité ne porte même que sur 4 bits (n). Les traitillés mettent en évidence les bits concernés par chaque opération.
4) à chaque soustraction, il faut pouvoir transmettre soit le nouveau reste, soit le reste précédent (restauration), selon qu'elle donne un emprunt de sortie nul ou non.

En visant une décomposition combinatoire, nous allons construire un soustracteur de 1 bit incorporant le sélecteur de sortie mentionné ci-dessus. Son schéma bloc est celui de la figure X.7.

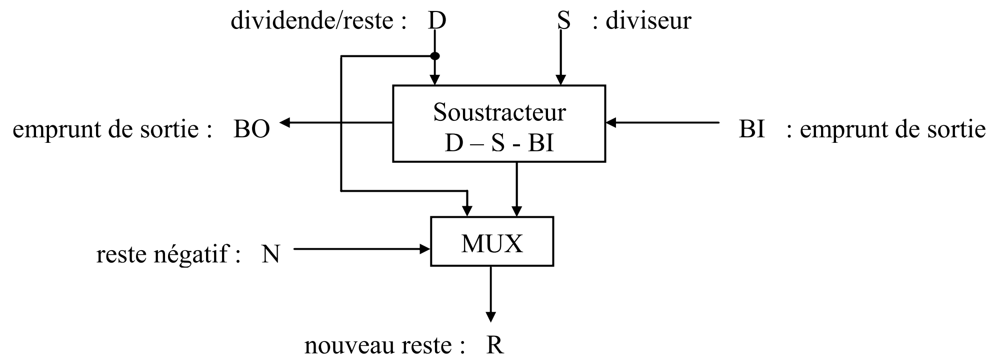

Fig. X.7

Les équations de sortie de ce module s'établissent facilement :

bit de reste $R = (D \text{ xor } S \text{ xor } BI) \text{ and } \overline{N} \text{ or } D \text{ and } N$

emprunt $BO = \overline{D} \text{ and } S \text{ or } \overline{D} \text{ and } BI \text{ or } S \text{ and } BI.$

Un tel module peut être câblé en réseau pour obtenir un diviseur d'un nombre quelconque de bits, en utilisant les enseignements 1), 2) et 3) ci-dessus. Pour diviser un nombre A de 8 bits par un nombre B de 4 bits et obtenir un quotient Q et un reste R de 4 bits, nous obtenons par exemple le réseau de la figure X.8.

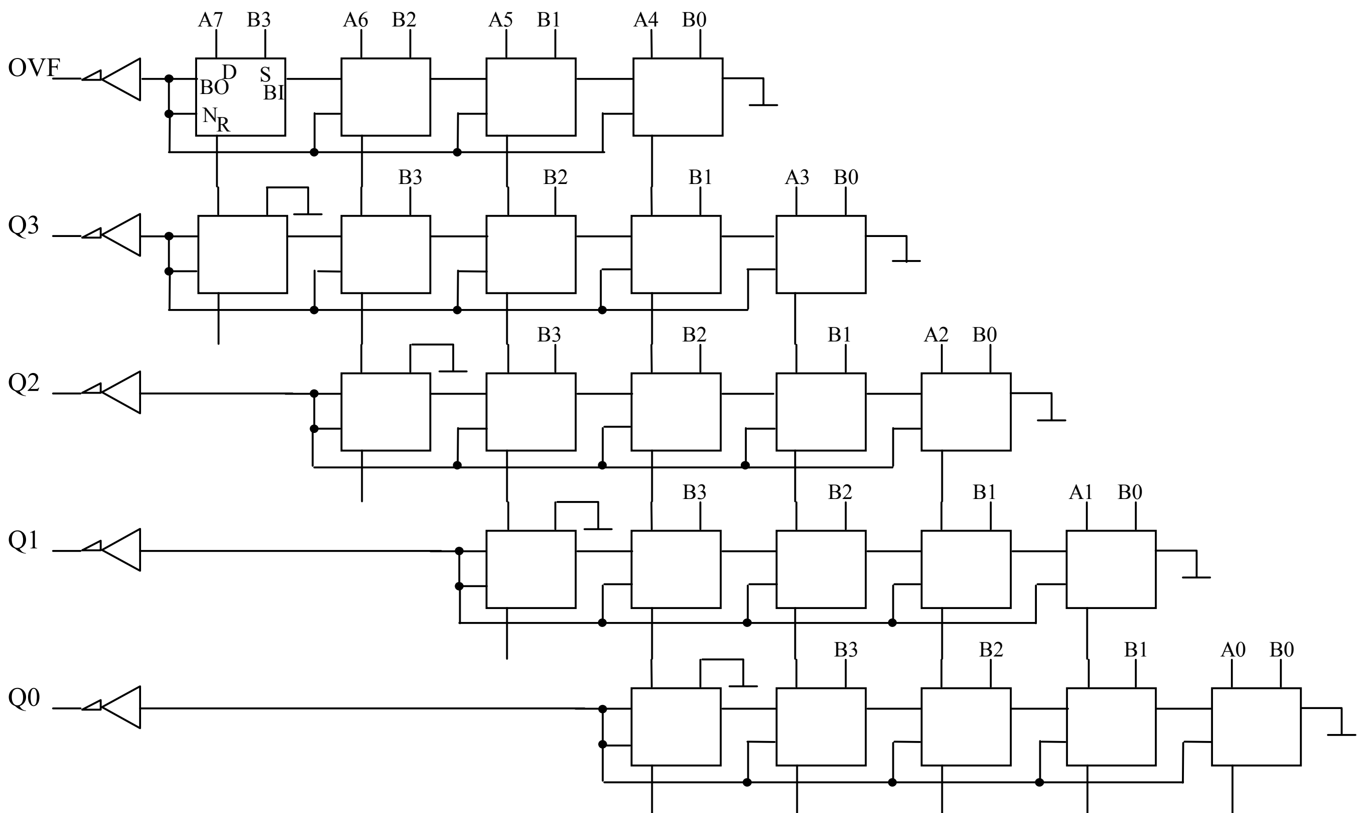

Fig. X.8

Bien entendu, rien ne nous empêche de créer des modules plus grands, effectuant par exemple des soustractions sur 4 bits, et même de faire un calcul anticipé de la retenue, tout en conservant une structure similaire à celle de la figure X.8. Malgré cela, la quantité de matériel va rapidement augmenter avec le nombre de bits. Pour une meilleure économie, il nous faut donc penser à une décomposition en séquence et à une réalisation sous la forme d'une MSS complexe.

Les observations faites sur la division à la main, mais aussi l'expérience acquise avec la multiplication, vont nous permettre d'établir un algorithme adapté à une réalisation séquentielle. Ainsi, comme pour la multiplication, nous allons créer une boucle dans laquelle un des opérandes perd un bit (le dividende) et le résultat en gagne un. Cela nous permettra d'économiser du temps et du matériel, en utilisant un registre de $2 \cdot n$ bits pour contenir le dividende au début du calcul, puis le reste et le quotient à la fin. Au lieu de décaler le diviseur sur la droite, nous allons pour cela décaler le dividende sur la gauche, afin d'obtenir l'alignement voulu à l'entrée du soustracteur.

Afin d'accélérer le calcul, nous établirons un algorithme qui ne restaure pas le reste lorsque le résultat de la soustraction est négatif. En effet, cette restauration revient à rajouter le diviseur au reste négatif obtenu, alors que nous allons de toutes façons décaler ensuite le reste à gauche (le multiplier par 2) et soustraire de nouveau le diviseur. Au lieu de cela, si le reste devient négatif (le bit correspondant du quotient est donc à 0), nous allons simplement continuer avec le prochain décalage mais remplacer la prochaine soustraction du dividende par une addition, puisque $(R + S) \cdot 2 - S = R \cdot 2 + S$. Toutefois, si le dernier reste ainsi obtenu est négatif (le dernier bit du quotient est nul), il faudra finir la restauration en lui ajoutant le diviseur afin d'obtenir un reste final correct.

Comme pour le diviseur câblé de la figure X.8, il nous faudra effectuer des soustractions sur n+1 bits. Mais en fait, nous pouvons utiliser un additionneur-soustracteur (la soustraction sera faite en additionnant le complément à 2) de n bits, plus quelques portes pour calculer le bit n+1.

Pour ce qui est des restes partiels, ce bit supplémentaire de poids fort nous donnera le signe. En effet, le résultat de chaque soustraction et de chaque addition restera compris entre DIVISEUR – 1 et –DIVISEUR. Puisque DIVISEUR s'exprime sans signe sur n bits, un bit supplémentaire permettra d'obtenir un résultat avec signe.

Appelons
- E : un bit supplémentaire à gauche du dividende
- N : le bit de signe du résultat de l'addition / soustraction
- C : le report de sortie de l'additionneur de n bits
- PQ : le précédent bit du quotient (soustraction si PQ = 1)
- NQ : le nouveau bit du quotient.

Nous avons $NQ = \overline{N} = E \text{ xor } C \text{ xor } \overline{PQ}$

soit $NQ = E \text{ xor } C$ pour une soustraction

et $NQ = E \text{ xor } \overline{C}$ pour une addition.

En utilisant le même flip-flop pour contenir alternativement E (décalé en série depuis le bit de poids fort du dividende_haut) et NQ (chargé en parallèle en même temps que le reste),

et en utilisant le bit de commande de la soustraction, SUB, en lieu et place de PQ dans le calcul de NQ, nous pouvons établir le schéma bloc simplifié de la figure X.9.

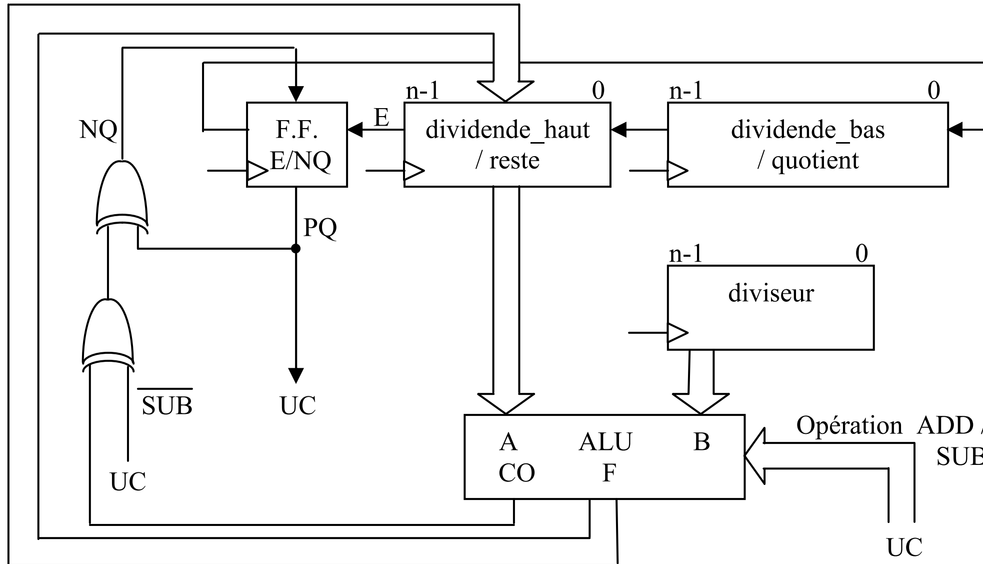

Fig. X.9

L'algorithme d'une division sans restauration utilisant l'UT ci-dessus peut être exprimé en pseudo Ada comme suit :

```
…
if diviseur=0 then goto erreur;   -- on ne divise pas par 0 !!
{else} E:= 0;                -- initialisation du flip-flop de signe
dividende_haut:= dividende_haut-diviseur;    -- E:=
ExorCxorSUB̅

if E=1 then goto depassement;          -- le quotient est ≥ 2^n
else
  for I in n-1 downto 0 loop
    if E=1 then
      decale_a_gauche(E ← dividende_haut
                              ← dividende_bas ← E);
      dividende_haut:= dividende_haut-diviseur;
                                          -- E:= ExorCxorSUB̅
  else
    decale_a_gauche(E ← dividende_haut
                              ← dividende_bas ← E);
    dividende_haut:= dividende_haut+diviseur;
                                          -- E:= ExorCxorSUB̅
  end if;
  end loop;
  if E=0 then
  -- si le dernier bit du quotient = 0, le reste est < 0
```

```
          decale_a_gauche(dividende_bas ← E);
                                            -- place le dernier bit de Q
            dividende_haut:= dividende_haut+diviseur;
                                                  -- corrige le reste
        else
          decale_a_gauche(dividende_bas ← E );
                              -- place le dernier bit de Q, mais ne corrige
                                     -- pas le reste car il est positif
        end if;
      end if;
      …
```

X.12 DIVISION D'ENTIERS EN COMPLÉMENT À 2

Adapter l'algorithme établi pour des nombres sans signe au traitement des nombres en complément à 2 n'est pas aussi simple pour la division que pour la multiplication. Nous nous contenterons de la démarche qui consiste à calculer séparément le signe et la valeur absolue du quotient.

Cette démarche peut être résumée de la façon suivante :

1) si le dividende est négatif, l'on prend son complément à 2
2) si le diviseur est négatif, l'on prend son complément à 2
3) l'on divise les nombres positifs ainsi obtenus comme s'il s'agissait de nombres sans signe
4) si le dividende était négatif, l'on prend le complément à 2 du reste
5) si le dividende et le diviseur étaient de signes différents, l'on prend le complément à 2 du quotient.

Divers algorithmes parviennent à éviter l'une ou l'autre de ces complémentations, mais leur étude sort du cadre de ce cours.

X.13 STRUCTURE D'UNE RALU

Comme nous l'avons vu déjà au chapitre VII, une unité arithmétique et logique ne permet d'effectuer de façon combinatoire que quelques opérations élémentaires. En ce qui concerne les opérations arithmétiques, cela se limite généralement à l'addition et à la soustraction. Pour réaliser des traitements plus complexes, il faudra les décomposer en une série d'opérations élémentaires. Cela implique que nous devrons conserver des résultats intermédiaires et qu'il sera donc pratique de disposer de quelques registres à cet effet, intégrés dans le même boîtier que l'ALU. D'où les ALUs à registres, ou RALUs (pour "Registered Arithmetic and Logic Unit").

Les figures X.5, X.6 et X.9 nous montrent que pour effectuer des multiplications et des divisions par addition / soustraction et décalage, il faut disposer au moins de 3 registres. L'un d'entre eux doit pouvoir fonctionner comme accumulateur. Une autre, que nous appellerons Q car il fournit le quotient lors d'une division, doit pouvoir être accouplé à l'accumulateur pour former un registre de $2 \cdot n$ bits. De plus, l'accumulateur et le registre Q doivent pouvoir être décalés en série, à gauche et à droite.

Dans une RALU, le décalage des registres ne se fait pas comme dans un '194 (registre à décalages universel, voir § VII.4) : plutôt que de charger le registre et de le décaler ensuite, il est plus rapide de charger directement la donnée décalée. Le décalage s'obtient au travers d'un multiplexeur appelé "barrel shifter" (décaleur en tonneau) placé à l'entrée du registre. La figure X.10 nous en montre le principe.

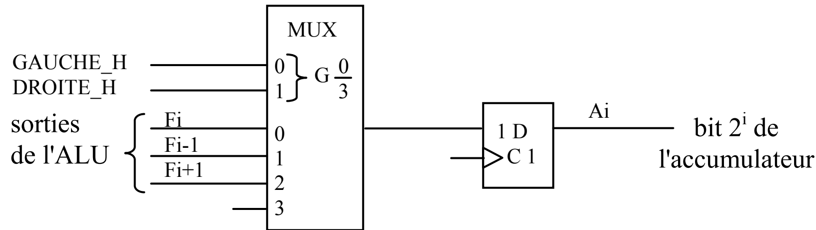

Fig. X.10

Le circuit Am 2901 de la maison Advanced Micro Devices (AMD) est un exemple typique de RALU standard. Bien que de conception relativement ancienne (il a été introduit sur le marché en 1976 avec d'autres circuits de la famille 2900), il reste le plus populaire, surtout sous la forme de diverses évolutions. Les figures X.11 et 12 nous montrent son schéma bloc.

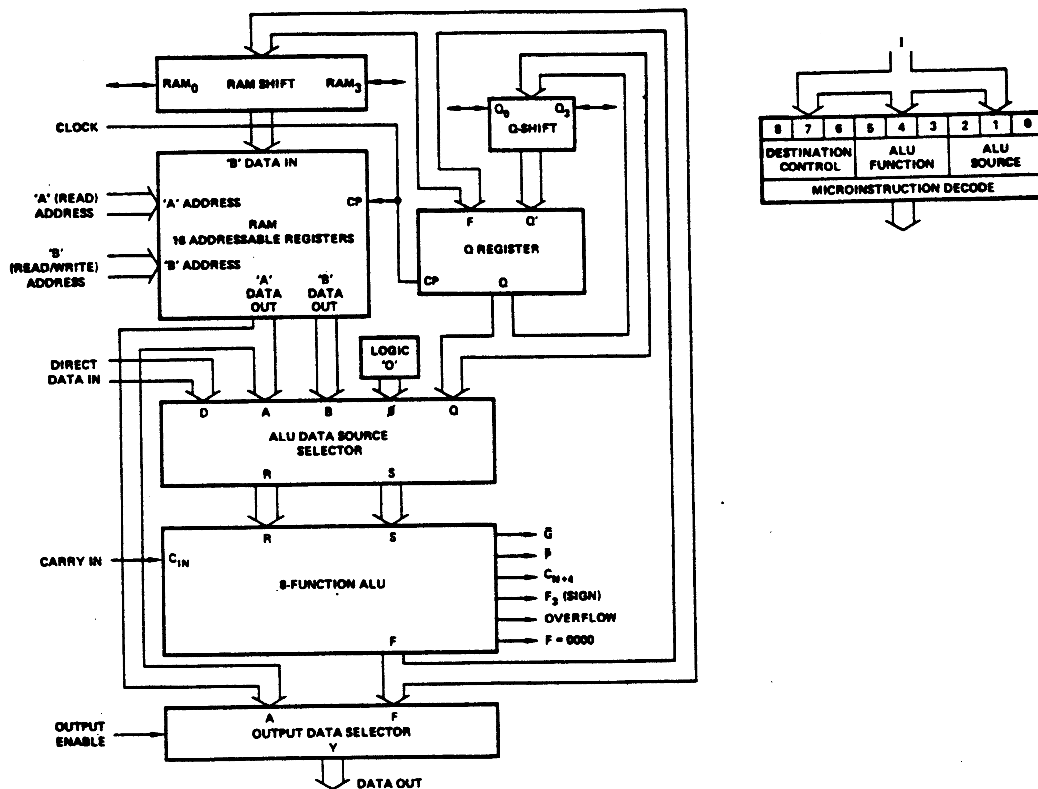

Fig. X.11

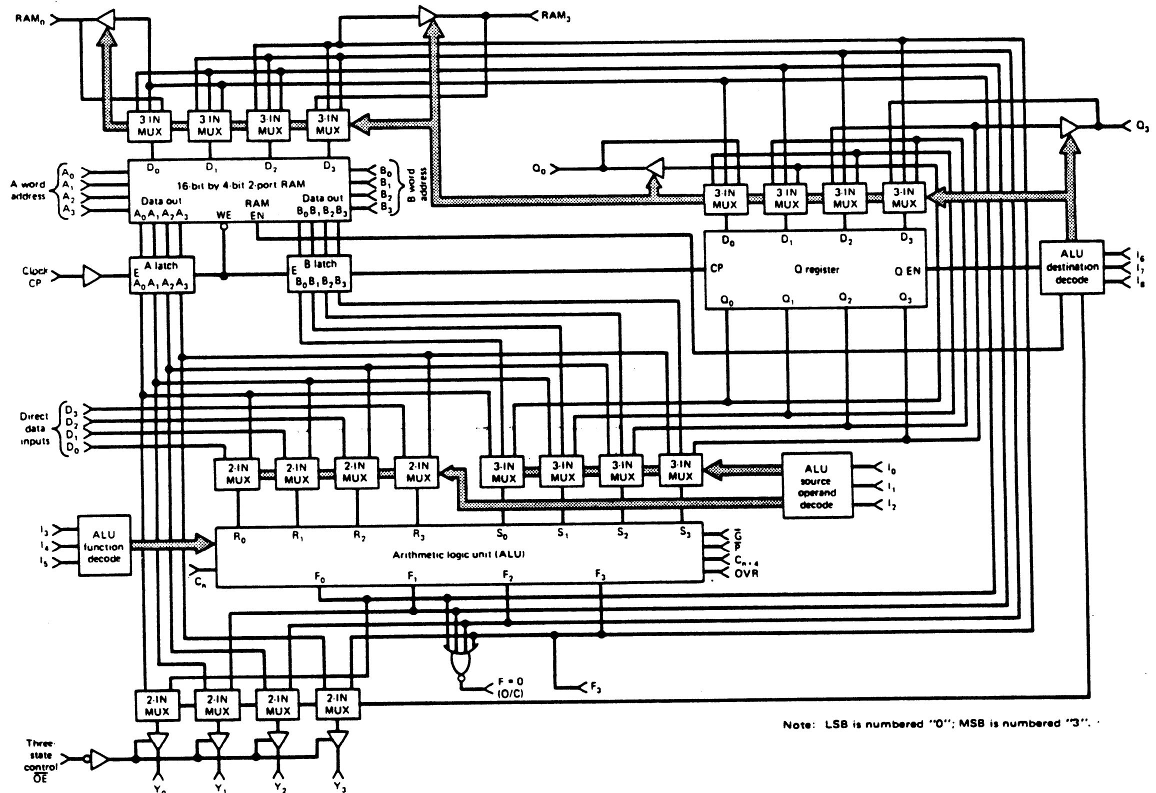

Fig. X.12

L'ALU du 2901 effectue 8 opérations qui apparaissent dans la table de la figure X.13 b). Elle comporte une entrée et une sortie de report, ainsi que les sorties G et P pour le calcul de retenue anticipé, ce qui nous permet de faire des opérations sur des nombres de k · 4 bits en connectant k circuits 2901 en parallèle. La figure X.14 nous montre la connexion de 4 2901s et d'un circuit de calcul de retenue anticipé (le 2902) pour créer une RALU de 16 bits.

ALU SOURCE OPERAND CONTROL

MNEMONIC	MICROCODE				ALU SOURCE OPERANDS	
	I_2	I_1	I_0	OCTAL CODE	R	S
AQ	L	L	L	0	A	Q
AB	L	L	H	1	A	B
ZQ	L	H	L	2	0	Q
ZB	L	H	H	3	0	B
ZA	H	L	L	4	0	A
DA	H	L	H	5	D	A
DQ	H	H	L	6	D	Q
DZ	H	H	H	7	D	0

Fig. X.13 a)

ALU FUNCTION CONTROL

MNEMONIC	MICROCODE				ALU FUNCTION	SYMBOL
	I_5	I_4	I_3	OCTAL CODE		
ADD	L	L	L	0	R Plus S	$R + S$
SUBR	L	L	H	1	S Minus R	$S - R$
SUBS	L	H	L	2	R Minus S	$R - S$
OR	L	H	H	3	R OR S	$R \vee S$
AND	H	L	L	4	R AND S	$R \wedge S$
NOTRS	H	L	H	5	$\bar{R}$ AND S	$\bar{R} \wedge S$
EXOR	H	H	L	6	R EX-OR S	$R \veebar S$
EXNOR	H	H	H	7	R EX-NOR S	$\overline{R \veebar S}$

Fig X.13 b)

ALU DESTINATION CONTROL

MNEMONIC	MICROCODE				RAM FUNCTION		Q REGISTER FUNCTION		Y OUTPUT	RAM SHIFTER		Q SHIFTER	
	I_8	I_7	I_6	OCTAL CODE	SHIFT	LOAD	SHIFT	LOAD		RAM_0	RAM_3	Q_0	Q_3
QREG	L	L	L	0	X	NONE	NONE	$F \rightarrow Q$	F	X	X	X	X
NOP	L	L	H	1	X	NONE	X	NONE	F	X	X	X	X
RAMA	L	H	L	2	NONE	$F \rightarrow B$	X	NONE	A	X	X	X	X
RAMF	L	H	H	3	NONE	$F \rightarrow B$	X	NONE	F	X	X	X	X
RAMQD	H	L	L	4	DOWN	$F/2 \rightarrow B$	DOWN	$Q/2 \rightarrow Q$	F	F_0	IN_3	Q_0	IN_3
RAMD	H	L	H	5	DOWN	$F/2 \rightarrow B$	X	NONE	F	F_0	IN_3	Q_0	X
RAMQU	H	H	L	6	UP	$2F \rightarrow B$	UP	$2Q \rightarrow Q$	F	IN_0	F_3	IN_0	Q_3
RAMU	H	H	H	7	UP	$2F \rightarrow B$	X	NONE	F	IN_0	F_3	X	Q_3

X = Don't Care. Electrically, the shift pin is a TTL input internally connected to a three-state output which is in the high-impedance state.
B = Register Addressed by B inputs.
UP is toward MSB; DOWN is toward LSB.

Fig. X.13 c)

ALU LOGIC MODE FUNCTIONS

OCTAL $I_{5,4,3}$, $I_{2,1,0}$	GROUP	FUNCTION
4 0	AND	$A \wedge Q$
4 1		$A \wedge B$
4 5		$D \wedge A$
4 6		$D \wedge Q$
3 0	OR	$A \vee Q$
3 1		$A \vee B$
3 5		$D \vee A$
3 6		$D \vee Q$
6 0	EX-OR	$A \veebar Q$
6 1		$A \veebar B$
6 5		$D \veebar A$
6 6		$D \veebar Q$
7 0	EX-NOR	$\overline{A \veebar Q}$
7 1		$\overline{A \veebar B}$
7 5		$\overline{D \veebar A}$
7 6		$\overline{D \veebar Q}$
7 2	INVERT	$\overline{Q}$
7 3		$\overline{B}$
7 4		$\overline{A}$
7 7		$\overline{D}$
6 2	PASS	Q
6 3		B
6 4		A
6 7		D
3 2	PASS	Q
3 3		B
3 4		A
3 7		D
4 2	"ZERO"	0
4 3		0
4 4		0
4 7		0
5 0	MASK	$\overline{A} \wedge Q$
5 1		$\overline{A} \wedge B$
5 5		$\overline{D} \wedge A$
5 6		$\overline{D} \wedge Q$

Fig. X.13 d)

ALU ARITHMETIC MODE FUNCTIONS

OCTAL $I_{5,4,3}$, $I_{2,1,0}$	C_n = L		C_n = H	
	GROUP	FUNCTION	GROUP	FUNCTION
0 0	ADD	A + Q	ADD plus one	A + Q + 1
0 1		A + B		A + B + 1
0 5		D + A		D + A + 1
0 6		D + Q		D + Q + 1
0 2	PASS	Q	Increment	Q + 1
0 3		B		B + 1
0 4		A		A + 1
0 7		D		D + 1
1 2	Decrement	Q - 1	PASS	Q
1 3		B - 1		B
1 4		A - 1		A
2 7		D - 1		D
2 2	1's Comp.	-Q - 1	2's Comp. (Negate)	-Q
2 3		-B - 1		-B
2 4		-A - 1		-A
1 7		-D - 1		-D
1 0	Subtract (1's Comp.)	Q - A - 1	Subtract (2's Comp.)	Q - A
1 1		B - A - 1		B - A
1 5		A - D - 1		A - D
1 6		Q - D - 1		Q - D
2 0		A - Q - 1		A - Q
2 1		A - B - 1		A - B
2 5		D - A - 1		D - A
2 6		D - Q - 1		D - Q

Fig X. 13 e)

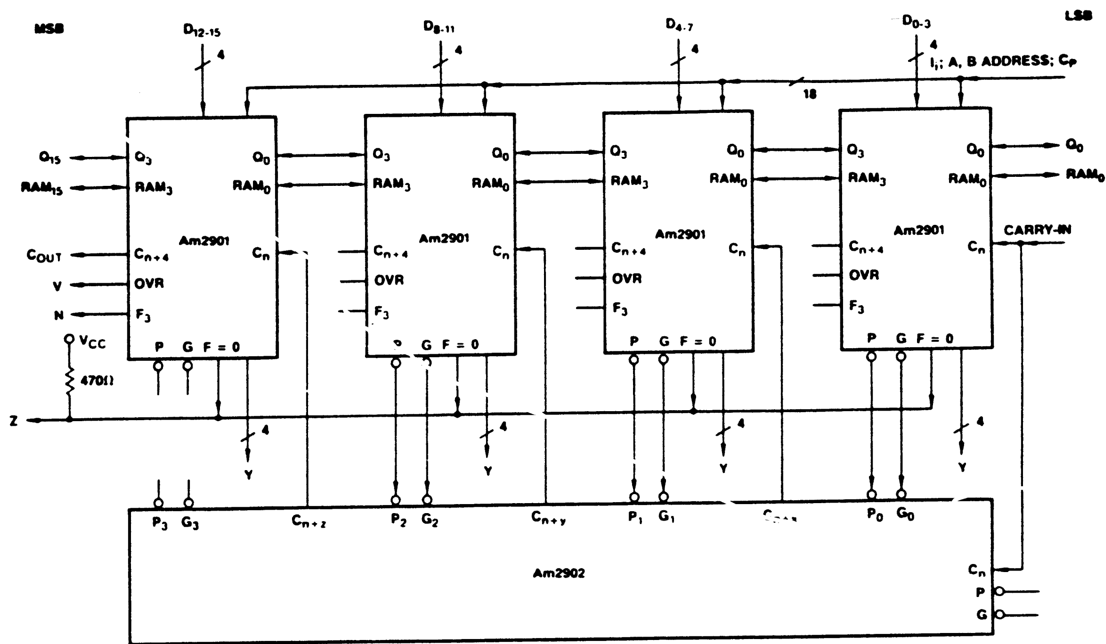

Fig. X.14

Le registre Q peut être chargé avec la sortie de l'ALU, sans décalage, ou recopié sur lui-même en décalant son contenu à travers le barrel shifter. Cela correspond bien aux besoins de la multiplication et de la division, puisque cela permet de l'initialiser avec le multiplicateur ou la partie basse du dividende, et d'effectuer ensuite les décalages.

Les autres registres, au nombre de 16, sont organisés de telle façon que l'on peut accéder aux sorties de deux d'entre eux et aux entrées de l'un d'entre eux simultanément. La sélection se fait à travers deux entrées d'adresse de 4 bits : l'adresse A et l'adresse B. L'adresse A sélectionne un registre parmi les 16, de façon à ce qu'il apparaisse sur les sorties A du bloc ('A' DATA OUT). L'adresse B sélectionne un registre parmi les 16 (cela peut d'ailleurs être le même que celui sélectionné par l'adresse A), de façon à ce qu'il apparaisse sur les sorties B ('B' DATA OUT). Mais en plus de cela, l'adresse B sélectionne le registre qui sera chargé avec le résultat de l'ALU (éventuellement décalé puisqu'il passe à travers le barrel shifter avant d'arriver vers les entrées 'B' DATA IN du bloc de registres). Dans une opération faisant intervenir deux registres de ce bloc, celui qui est désigné par l'adresse B pourra donc fonctionner comme accumulateur (contient un opérande au début de l'opération, et le résultat à la fin).

La structure interne de ce bloc de registres est schématisé à la figure X.15. Il s'agit d'une mémoire à lecture-écriture (connue sous le sigle "RAM", pour Random Access Memory, ou mémoire à accès aléatoire) à double accès. En fait seule la lecture est à double accès simultané, puisque cette mémoire possède 2 sorties de 4 bits, chacune permettant d'accéder (lire) à un mot mémorisé.

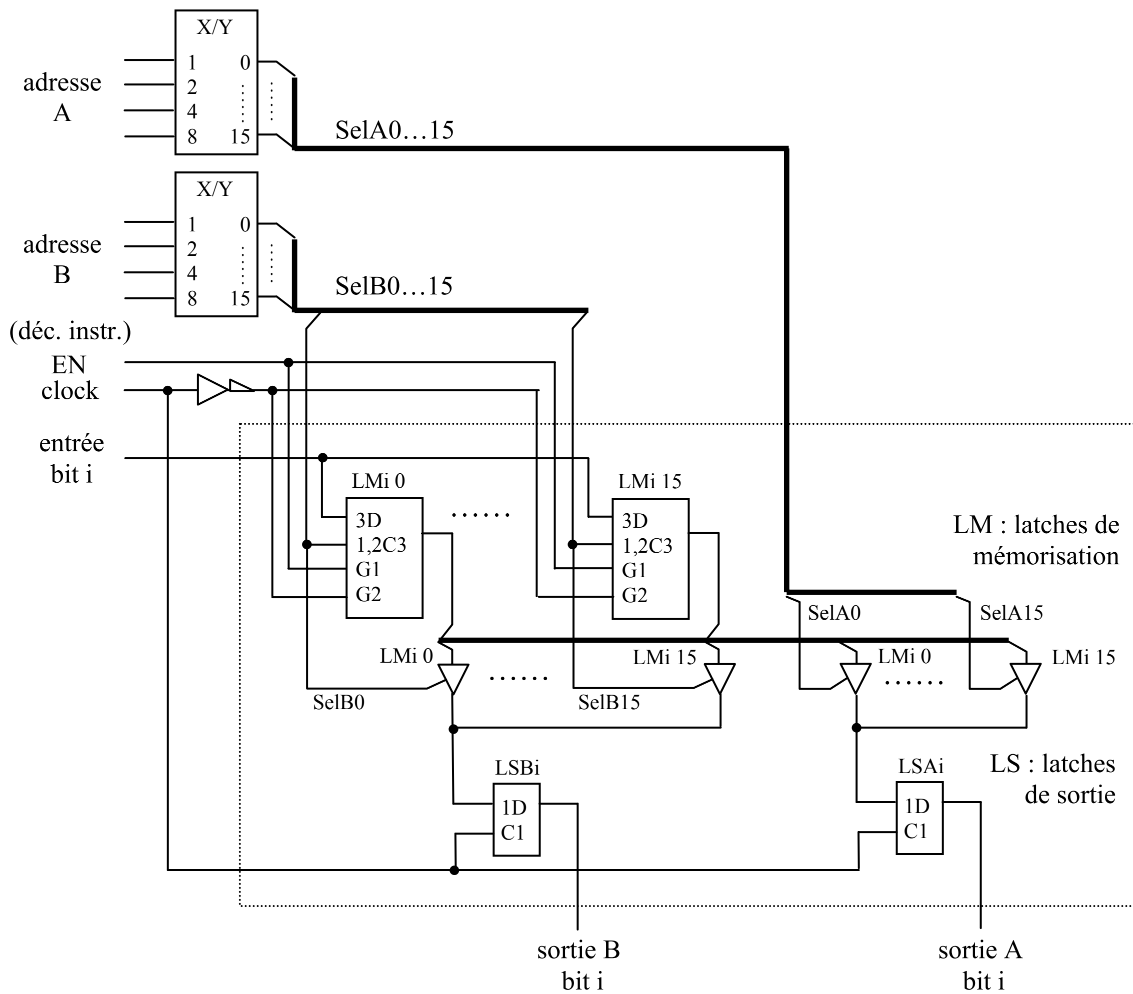

Fig. X.15

La mémorisation se fait dans des latches, marqués LMi 0..15 dans le schéma ci-dessus. Mais les sorties passent à travers des latches marqués LSBi et LSAi, qui sont ouverts en opposition avec ceux de la mémorisation, rendant ainsi l'ensemble non transparent (nous obtenons des bascules maître-esclave). L'indice i indique le rang du bit dans le mot. Pour les 4 bits de large d'un 2901, la partie entourée d'un traitillé est donc reproduite 4 fois.

Les opérandes R et S de l'ALU sont sélectionnés à travers des MUX, parmi l'entrée directe D, la sortie A et la sortie B de la RAM, le registre Q et la valeur fixe zéro. La table de la figure X.13 a) et le schéma de la figure X.12 nous montre que seules 8 combinaisons sont possibles.

La table X.13 c) nous montre les diverses commandes agissant sur l'écriture dans la RAM et le registre Q, les décalages avec les 2 barrel shifters et la sélection de ce que le 2901 fournit sur les sorties Y. Les tables X.13 d) et e) nous montrent les différentes combinaisons possibles entre opérandes et opérations.

Nous reviendrons sur ce circuit au chapitre XI, car nous pouvons l'utiliser pour réaliser un processeur. L'utilité de certaines de ses particularités deviendra alors plus claire.

X.14 VIRGULE FIXE ET VIRGULE FLOTTANTE

Dans ce qui précède, nous nous sommes contentés d'étudier les opérations arithmétiques sur des nombres entiers. En fait, les circuits que nous avons vus peuvent être utilisés pour effectuer des calculs sur des nombres réels représentés de la façon habituelle avec une virgule séparant la partie entière de la partie fractionnaire. Il suffit de ne pas représenter formellement la virgule, ce qui est possible si on lui attribue une place fixe. D'où le nom de représentation en virgule fixe.

Avec N bits pour représenter un nombre sans signe, le rapport entre le nombre le plus grand et le nombre le plus petit non nul, en virgule fixe, est égal à $2^N - 1$. Le nombre total de valeurs différentes est bien sûr égal à 2^N.

Travailler à la fois avec des nombres très grands et des nombres très petits en virgule fixe demande soit un nombre N de bits très grand, soit la prise en compte d'un facteur d'échelle (ou d'un ordre de grandeur) variable. Pour des raisons économiques, c'est cette dernière option qui est généralement choisie : il est tout de même plus pratique de travailler avec notre calculette en notation scientifique, plutôt qu'avec une calculette qui aurait besoin d'un affichage d'un mètre de long !

La notation en virgule flottante n'est rien d'autre qu'une notation scientifique en binaire. Notre nombre est décomposé en deux parties : d'une part un certain nombre de chiffres significatifs, constituant ce que l'on appelle la mantisse, et d'autre part un facteur d'échelle qui est une puissance entière de la base et qui est appelé exposant. La mantisse et l'exposant sont représentés côte à côte dans les N bits à disposition.

Prenons un exemple avec des représentations sur 16 bits. Le nombre le plus grand représentable en virgule fixe, en mettant la virgule à droite du bit de poids faible, est $2^{16} - 1$ soit 65'535. Pour une représentation en virgule flottante, attribuons 6 bits à la représentation de l'exposant et les 10 autres à la représentation de la mantisse. Le plus grand exposant sera $2^6 - 1 = 63$, donc le plus grand facteur d'échelle sera 2^{63}. La plus grande mantisse sera $2^{10} - 1 = 1'023$. Donc le plus grand nombre représentable dans cette notation sera $1'023 \cdot 2^{63}$, soit plus de $9,4 \cdot 10^{21}$.

Le rapport entre le plus grand nombre et le plus petit non nul (ce que l'on appelle la dynamique d'une représentation) est spectaculairement meilleur en virgule flottante qu'en virgule fixe. Mais puisque le nombre total de valeurs différentes représentables reste égal à 2^N, ce gain en dynamique se fait au détriment de la précision absolue. En effet, dans l'exemple ci-dessus l'erreur d'arrondi en virgule fixe est de ± 0,5, mais peut aller jusqu'à $\pm 0,5 \cdot 2^{63}$ dans la représentation en virgule flottante que nous avons choisi (soit ½ LSBit de la mantisse, fois le facteur d'échelle). Par contre, la précision relative qui peut être très faible en virgule fixe (± 50 % pour des valeurs entre 1 et 2 si nous ne représentons que des entiers), peut être maintenue dans des limites raisonnables en virgule flottante si la mantisse est choisie de façon à ce que son bit de poids fort soit à 1. Pour notre exemple de notation ci-dessus, nous aurions une erreur relative maximale de ± 0,5 / 512, soit environ 1‰ dans toute la gamme des valeurs représentables.

X.15 FORMATS STANDARD EN VIRGULE FLOTTANTE

Un format de représentation en virgule flottante est un ensemble de règles définissant la façon de coder un nombre. Ces règles définissent donc le nombre de bits attribués respectivement à la mantisse et à l'exposant, leur position relative, leur notation, et d'autres particularités.

Chaque grand fabricant d'ordinateurs à créé son ou ses formats "standard", mais nous nous limiterons ici à étudier les formats proposés par l'IEEE (Institute of Electrical and Electronics Engineers), qui ont été adoptés par les fabricants de microprocesseurs.

L'IEEE propose trois formats de base :

- un format simple précision, utilisant 32 bits
- un format double précision, utilisant 64 bits
- un format quadruple précision, utilisant 128 bits.

La mantisse est un nombre fractionnaire en virgule fixe dont la partie entière vaut 1. Cette convention permet de rendre unique la représentation d'un nombre dans un format donné, et de minimiser l'erreur relative d'arrondi. Une mantisse obéissant à cette convention est dite "normalisée".

Puisque la partie entière de la mantisse dans une représentation normalisée est toujours égale à 1, il n'est pas nécessaire de la représenter explicitement dans le nombre codé. Lorsque ce bit n'est pas représenté, ce qui est le cas dans les formats simple et double précision IEEE, nous parlerons de bit caché ("hidden bit").

Un bit est utilisé pour indiquer le signe du nombre (0 = positif, 1 = négatif), la mantisse représentant la valeur absolue (au facteur d'échelle près).

L'exposant est un nombre signé représenté dans une notation biaisée (excédent de …).

La norme IEEE prévoit également des codages spéciaux pour des valeurs particulières :

- le zéro n'admet pas de représentation normalisée; il n'est pas non plus pratique de le représenter par la plus petite valeur positive (ou négative) normalisable, car nous aurions alors $X + \text{zéro} \neq X$ et $X \cdot \text{zéro} \neq \text{zéro}$. La norme prévoit de coder le zéro par une valeur d'exposant et une valeur de mantisse toutes deux nulles, et un signe quelconque. Il s'agit bien d'un exposant représenté par des zéros dans une notation en excédent de $2^{n-1} - 1$, donc de la valeur la plus petite du facteur d'échelle (valeur maximale négative de l'exposant).
- les infinis sont notés avec un exposant dont tous les bits sont à 1 (ce qui correspond à sa valeur maximale positive) et une mantisse nulle.
- certains codes sont prévus pour la représentation d'erreurs ou de toute autre information inventée par l'utilisateur; ces informations ne sont pas des nombres, d'où la terminologie "Not a Number" (NaN). C'est de nouveau la valeur maximale positive de l'exposant qui est réservée à cet effet, mais avec une mantisse non nulle.

N.B. Les NaNs servent notamment à propager convenablement les erreurs lors des calculs: lorsqu'une opération est impossible, par exemple $X / 0$ ou encore $(+\infty) - (-\infty)$, le résultat rendu est un NaN réservé à cet effet. Une opération dont l'un des opérandes est un NaN donne un NaN en résultat. Ceci permet d'évaluer une expression

complexe sans se soucier de la validité de chaque opération élémentaire : si une impossibilité se manifeste en cours d'évaluation, le résultat final sera un NaN.

La table de la figure X.16 résume les interprétations d'un nombre en virgule flottante selon la norme IEEE simple précision, alors que la figure X.17 illustre son format. Les notations abrégées utilisées dans la table sont les suivantes :

le nombre en virgule flottante F = (S, E, M)

où S = le signe du nombre

E = la représentation de l'exposant en excédent de 127 (= $2^{8-1} - 1$)

M = la représentation de la mantisse (à laquelle il manque le bit caché)

l'exposant e = E – 127

la partie fractionnaire de la mantisse $mf = \frac{M}{2^{23}}$

forme de F	interprétation	
$-126 \leq e < 127$	nb. normalisé	$(-1)^S (1 + mf) 2^e$
$e = -127, mf \neq 0$	nb. dénormalisé	$(-1)^S (mf) 2^{-126}$
$e = -127, mf = 0$	zéros	$(-1)^S 0$
$e = +128, mf = 0$	infinis	$(-1)^S \infty$
$e = +128, mf \neq 0$	pas des nombres	NaN

Fig. X.16

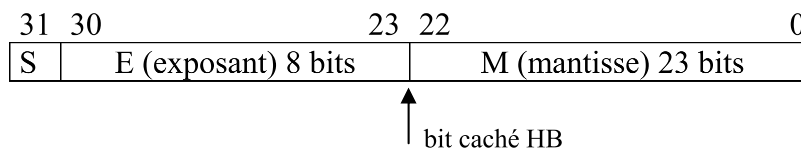

Fig. X.17

La norme IEEE prévoit également deux formats dérivés des formats de base et appelés <u>formats étendus</u> à simple et à double précision. Les cinq formats sont résumés dans la table de la figure X.18.

A titre d'exemple, voici la représentation du nombre $1.225 \cdot 10^3$ (soit 10011001001 en base 2) dans les formats IEEE

simple précision : 0 1000 1001 0011 0010 0100 0000 0000 000

simple précision étendu : 0 100 0000 1001 1001 1001 0010 0000 0000 0000 0000 0000

Dans le format simple précision, le bit de poids fort (partie entière) de la mantisse n'est pas représenté : il est caché. Il est par contre représenté dans le format simple précision étendu.

Voici un second exemple, avec la représentation du nombre $1.225 \cdot 10^2$ (soit 1100,01 en base 2) dans les formats IEEE

simple précision : 0 1000 0010 1000 1000 0000 0000 0000 000

Tableau récapitulatif des différents formats IEEE

formats IEEE nb. de bits	simple précision 32	simple précision étendu 44	double précision 64	double précision étendu 80	quadruple précision 128
nb. bits E	8	11	11	15	15
notation E	excédent 127	excédent 1'023	excédent 1'023	excédent 16'383	excédent 16'383
val. min. max. e	-126, +127	-1'022, +1'023	-1'022, +1'023	-16'382, +16'383	-16'382, +16'383
nb. bits M	23	32	52	64	112
bit caché	oui	non	oui	non	Non
val. max. F	$(2 - 2^{-23}) \cdot 2^{127}$	$(2 - 2^{-31}) \cdot 2^{1'023}$	$(2 - 2^{-52}) \cdot$ 1'023	$(2 - 2^{-63}) \cdot 2^{16'383}$	$(2 - 2^{-111}) \cdot 2^{16'383}$
val. min. F ≠ 0	2^{-150}	$2^{-1'053}$	$2^{-1'074}$	$2^{-16'445}$	$2^{-16'493}$

Fig. X.18

X.16 LES OPÉRATIONS SUR LES NOMBRES EN VIRGULE FLOTTANTE

Les opérations d'addition, soustraction, multiplication et division en virgule flottante peuvent être effectuées à l'aide d'une ALU prévue pour le calcul en virgule fixe, mais demandent plus de manipulations. La table de la figure X.19 résume la façon de procéder, avec les notations abrégées suivantes :

le nombre A est égal à $m_A \cdot 2^{e_A}$ $(B = m_B \cdot 2^{e_B})$

où m_A est la mantisse, y compris le signe et l'éventuel bit caché

et e_A est l'exposant du nombre A.

$A + B = (m_A \cdot 2^{e_A - e_B} + m_B) \cdot 2^{e_B}$	si $e_A \leq e_B$
$A + B = (m_B \cdot 2^{e_B - e_A} + m_A) \cdot 2^{e_A}$	si $e_A \geq e_B$
$A - B = (m_A \cdot 2^{e_A - e_B} - m_B) \cdot 2^{e_B}$	si $e_A \leq e_B$
$A - B = (-m_B \cdot 2^{e_B - e_A} + m_A) \cdot 2^{e_A}$	si $e_A \geq e_B$
$A \times B = (m_A \cdot m_B) \cdot 2^{e_A + e_B}$	
$A \div B = (m_A \div m_B) \cdot 2^{e_A - e_B}$	

Fig. X.19

Pour l'addition et la soustraction, il faut s'assurer que les deux nombres ont le même facteur d'échelle. Si ce n'est pas le cas, il faut dé-normaliser le plus petit, en divisant m par 2 et en ajoutant 1 à son exposant, jusqu'à ce que les deux exposants soient égaux.

Les quatre opérations peuvent fournir un résultat dé-normalisé. Il faudra donc les terminer par une équation de normalisation. La normalisation se fait en décalant la mantisse du résultat à droite et en incrémentant son exposant, ou en décalant la mantisse à gauche et en décrémentant l'exposant, jusqu'à ce que la partie entière de la mantisse soit égale à 1.

La figure X.20 nous montre un organigramme d'un sous-microprogramme (l'exécution se termine par un retour au microprogramme appelant) effectuant une addition C = A + B ou une soustraction C = A – B de nombres en virgule flottante. Les notations sont les mêmes que pour la figure X.19.

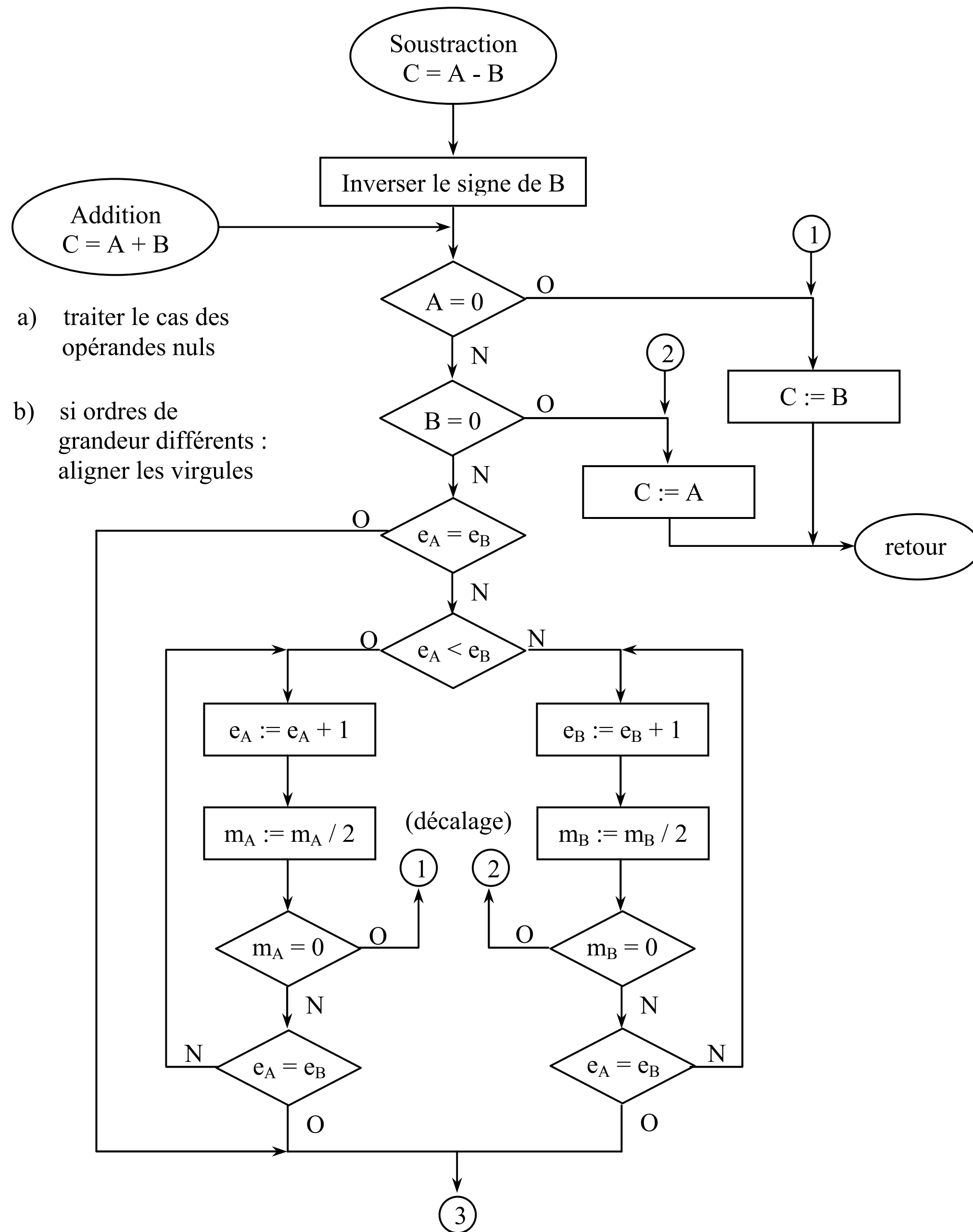

Fig. X.20

c) effectuer l'addition

d) traiter le cas du résultat nul

e) normaliser le résultat si cela est possible

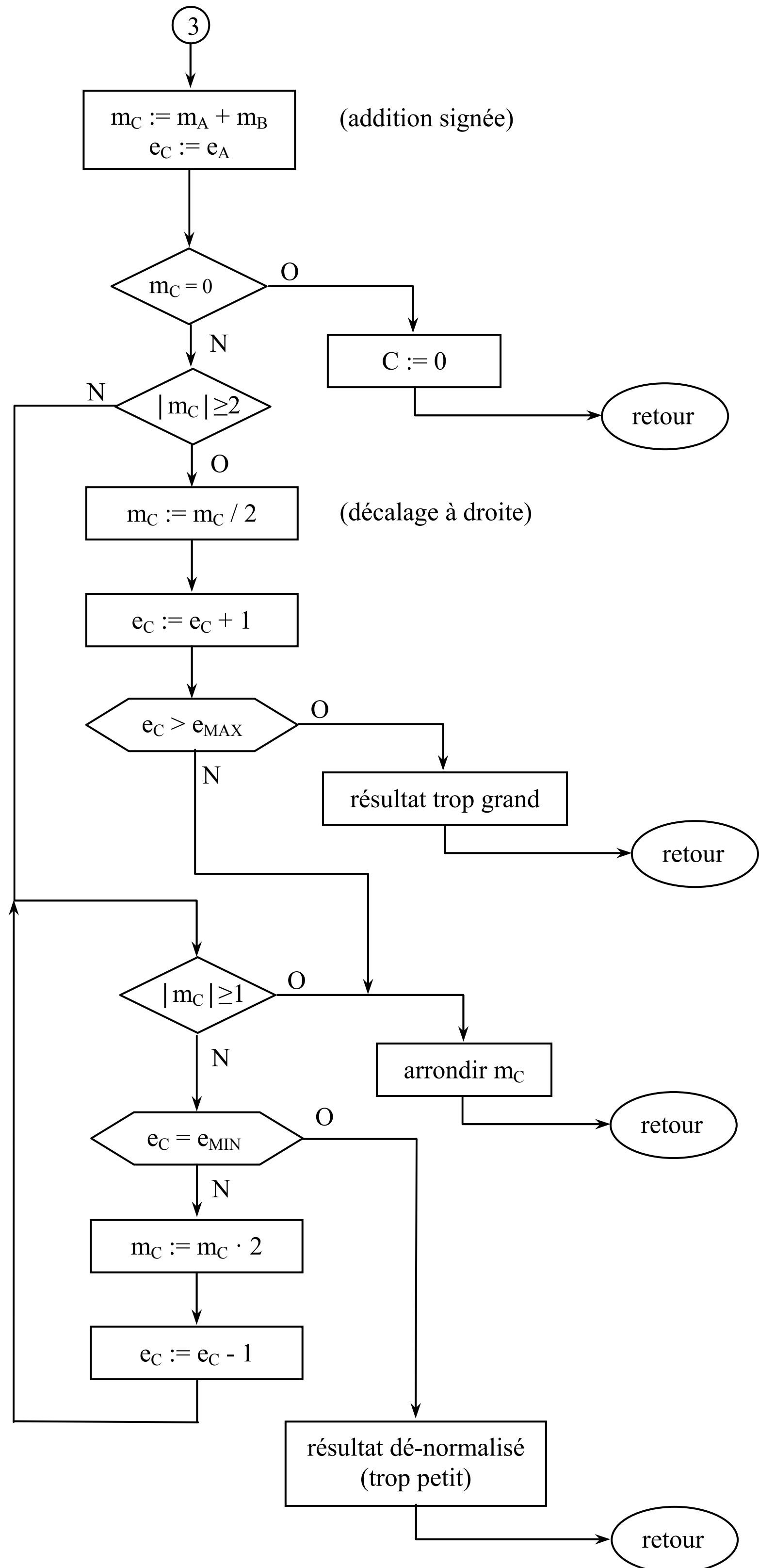

Fig. X.20 (suite)

Les organigrammes des figures X.21 et X.22 nous montrent le déroulement d'une multiplication et d'une division en virgule flottante, respectivement. La normalisation du résultat n'y est pas détaillée, car elle est identique à ce que nous avons vu pour l'addition et la soustraction.

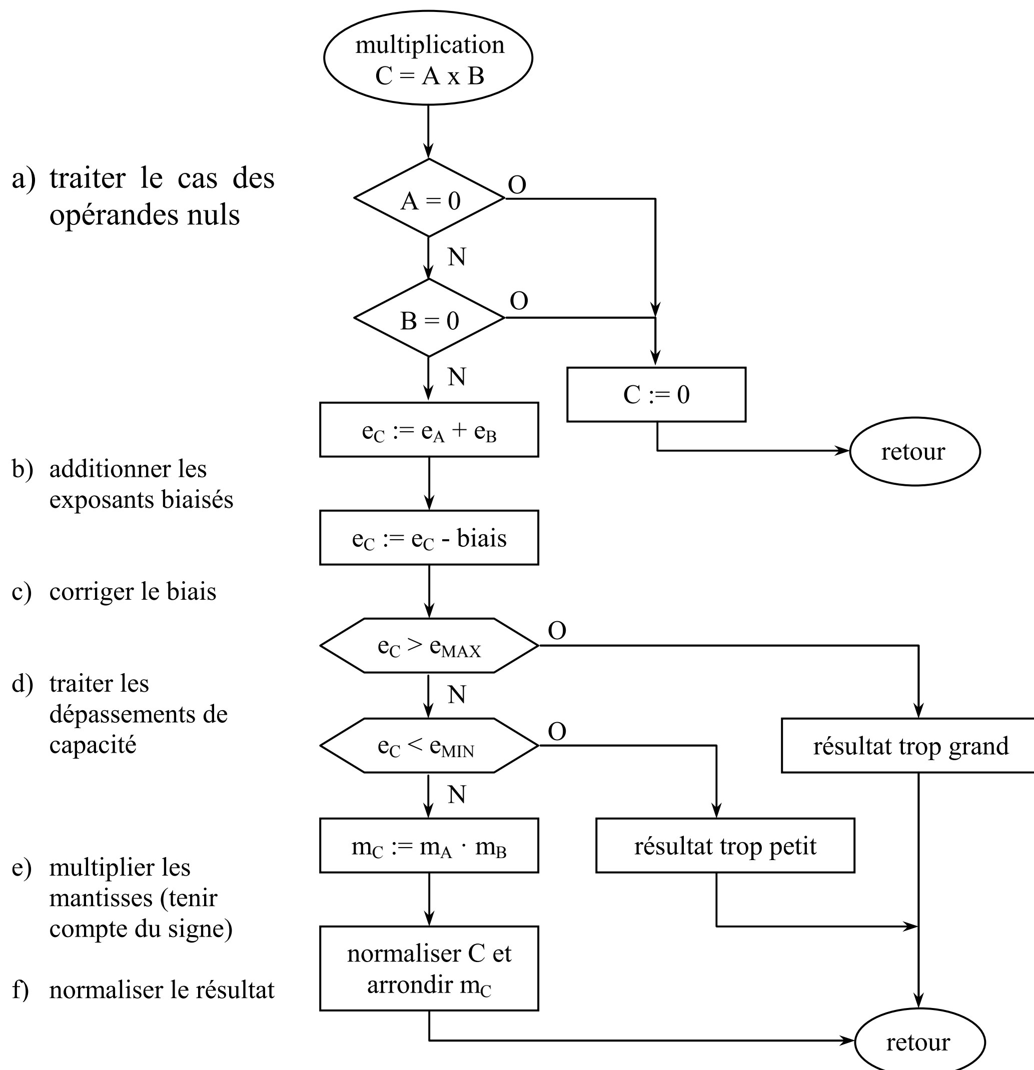

Fig. X.21

En additionnant deux nombres biaisés, nous obtenons un résultat avec un biais doublé. Il faut donc corriger le résultat en lui soustrayant une fois le biais. Par exemple : en additionnant 2 nombres notés en excédent de 127, nous obtenons un résultat noté en excédent de 254, pour le convertir en une notation en excédent de 127, il faut donc bien lui soustraire 127.

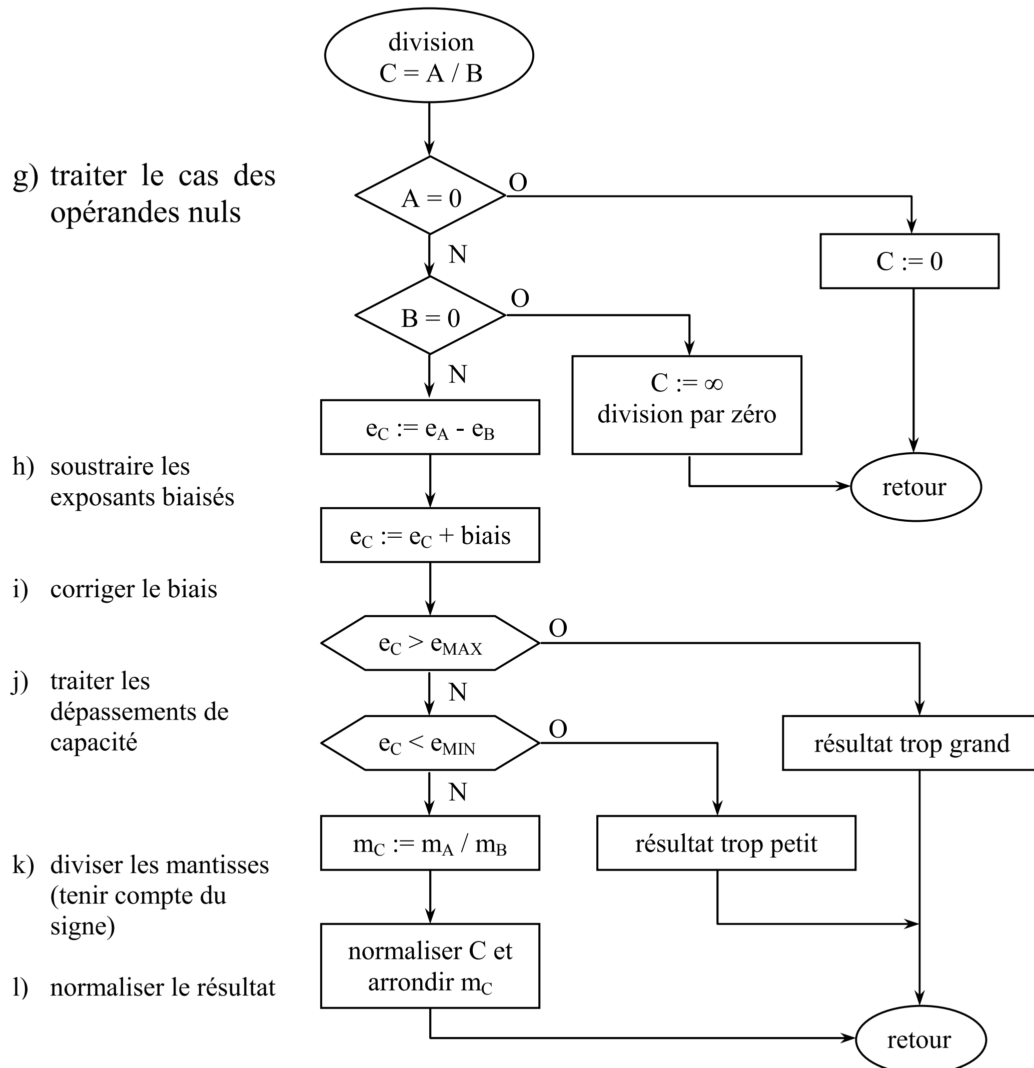

Fig. X.22

En plus d'un résultat, les opérations (plus précisément : les circuits effectuant des opérations sur les nombres flottants) fournissent quelques variables d'état ou <u>indicateurs</u> ("status bits", "flags"). Ce seront typiquement les indicateurs suivants :

- résultat inexact : le résultat fourni n'est pas le résultat exact car il a dû être arrondi;
- résultat trop grand : ("overflow") la valeur absolue du résultat est trop grande pour être représentable dans le format considéré;
- résultat trop petit : ("underflow") la valeur absolue de résultat est trop petite pour être représentable sous forme normalisée dans le format considéré;
- division par zéro : indique que l'on a essayé de diviser un nombre non nul par zéro;
- opération invalide : ce sont les opérations $\infty - \infty$, $0 \cdot \infty$, $0 / 0$, ∞ / ∞, ainsi que les opérations où un des opérandes (ou les deux) est un NaN;
- opérande dénormalisé : l'un des opérandes est un nombre dénormalisé.

X.17 BITS DE GARDE ET ARRONDIS

Pour effectuer des calculs sur des nombres flottants à l'aide d'une RALU, la mantisse et l'exposant de chacun des opérandes sont chargés dans des registres internes. On attribue généralement aux mantisses des registres ayant une taille supérieure à celle d'une mantisse (y compris le bit caché). La mantisse est chargée justifiée à gauche, donc la partie entière de la mantisse correspondant au bit le plus significatif du registre, et les bits supplémentaires situés à droite du registre sont chargés à zéro. Ces bits supplémentaires sont appelés "bits de garde". Leur raison d'être est mise en évidence par l'exemple ci-dessous.

Prenons deux nombres au format IEEE simple précision ayant des valeurs très proches. Par exemple A = $1{,}00\ldots00 \cdot 2^1$ et B = $1{,}11\ldots11 \cdot 2^0$. Pour effectuer la soustraction A – B, la mantisse du nombre B doit être décalée d'un rang à droite afin d'aligner les virgules. Dans ce décalage, B perd le bit le moins significatif de sa mantisse.

$$
\begin{array}{rcl}
A & = & 1,00\ldots00 \cdot 2^1 \\
-B & = & 0,11\ldots11 \cdot 2^1 \\
\hline
C & = & 0,00\ldots01 \cdot 2^1
\end{array}
$$

après normalisation : $C = 1,00\ldots00 \cdot 2^{-22}$

Effectuons la même opération avec 4 bits de garde. Cette fois-ci, le bit le moins significatif de la mantisse de B n'est pas perdu lors du décalage à droite : il va dans le bit de garde le plus significatif.

$$
\begin{array}{rcl}
A & = & 1,00\ldots00\ \ 0000 \cdot 2^1 \\
-B & = & 0,11\ldots11\ \ 1000 \cdot 2^1 \\
\hline
C & = & 0,00\ldots00\ \ 1000 \cdot 2^1
\end{array}
$$

après normalisation : $C = 1,00\ldots00\ \ \underbrace{0000}_{\text{bits de garde}} \cdot 2^{-23}$

Les bits de garde permettent donc d'améliorer la précision du calcul, particulièrement lorsque la mantisse dé-normalisée du résultat est très petite.

Les bits de garde ne sont utilisés que pendant le calcul. Pour fournir un résultat au format standard, il s'agira d'arrondir la mantisse. Il existe quatre options possibles d'arrondi :

- arrondi vers zéro : (ou troncature), le résultat rendu correspond au nombre représentable de valeur absolue immédiatement inférieure à celle du résultat réel; c'est la façon de procéder la plus simple puisqu'il suffit d'ignorer simplement les bits de garde du résultat;
- arrondi vers +∞ : (respectivement vers -∞), le résultat rendu correspond au nombre représentable le plus proche supérieur (respectivement inférieur) au résultat réel;
- arrondi au plus près : le résultat correspond au nombre représentable le plus proche du résultat réel (valeur absolue immédiatement supérieure si le bit de garde le plus significatif du résultat réel vaut 1, immédiatement inférieure dans le cas contraire).

X.18 EXERCICES

X.18.1 Établir le schéma d'un convertisseur purement combinatoire BCD à binaire utilisant des additionneurs 4 bits de type 'LS283, pour une entrée BCD de 3 chiffres représentant un entier entre 0 et 999.

X.18.2 Établir un organigramme pour effectuer la conversion d'un entier BCD de 4 chiffres (0 à 9999) en binaire, à l'aide d'une UT ne comportant qu'une seule RALU de type 2901 (un seul IC de type 2901).

X.18.3 Établir le schéma bloc d'une UT et l'organigramme de fonctionnement d'un convertisseur binaire – BCD travaillant selon l'algorithme suivant :

a) le nombre binaire est mis dans un registre BINREG
b) un registre BCDREG est attribué au futur nombre BCD, et initialisé à 0
c) additionner 3 à chaque décade du registre BCD qui excède 0100
d) décaler les deux registres à gauche (une position), BCDREG ← BINREG
e) répéter les étapes 3) et 4) autant de fois qu'il y a de bits dans le registre BIREG.

N.B. BCDREG et BINREG sont reliés en série, avec BCDREG du côté poids fort.

X.18.4 Établir le schéma d'un circuit de multiplication purement combinatoire, pour multiplier deux entiers binaires sans signe de 4 bits selon la méthode utilisée pour calculer à la main.

X.18.5 Établir le schéma d'une UT basée sur des 2901 et le microprogramme d'une UC basée sur MISEQv3, pour multiplier des nombres de 16 bits en complément à 2. Que manque-t-il à MISEQv3 pour faciliter cette réalisation ?

X.18.6 L'algorithme de multiplication de Booth est basé sur la constatation suivante : une suite d'additions du multiplicande, chaque fois décalé d'un rang supplémentaire, peut-être remplacé par une soustraction et une addition. Par exemple : Mcande · 1111 = Mcande · 10000 – Mcande.

Établir un algorithme en pseudo-Ada qui tire profit de la constatation ci-dessus, et le vérifier en multipliant les nombres en complément à 2 suivants : 0110 1010 · 1110 0111.

Indice : examiner les 2 derniers bits du multiplicateur, à chaque décalage, pour repérer le début et la fin d'une suite de 1.

X.18.7 Établir le schéma d'un diviseur purement combinatoire utilisant des additionneurs 4 bits de type 'LS283, pour effectuer des divisions de nombres de 16 bits par des nombres de 8 bits, sans signe.

X.18.8 Vérifier l'algorithme des pages 19-20 en l'appliquant à la main pour diviser 185 par 13 et 50 par 10 (8 bits / 4 bits).

X.18.9 Établir en pseudo-Ada un algorithme de division avec restauration pour des nombres sans signe.

X.18.10 Établir le schéma d'une UT basée sur des 2901 et le microprogramme d'une UC basée sur MISEQv4 (voir X.18.5), pour diviser un nombre de 16 bits par un nombre de 8 bits en complément à 2.