Les DSP ... G. Baudoin / F. Virolleau - Dunod

Photocopie : Conclusion.

p 251 - 255 p 12 - p 32

p 273 - 295

Programmation en C.

Intérêts et inconvénients -

Ecrire un programme en langage C est plus simple et plus rapide qu'écrire le même algorithme en assembleur. De plus, l'écriture en C garantit la portabilité du programme sur différents processeurs.

Malheureusement, le programme assembleur généré par le compilateur C à partir du fichier source en langage C est beaucoup moins efficace, à la fois en termes de vitesse d'exécution et en terme d'occupation mémoire, que le même programme écrit directement en assembleur par un programmeur connaissant bien l'architecture du DSP cible. Toutefois, l'optimiseur C permet de diminuer cette différence d'efficacité.

D'autres caractéristiques du compilateur C permettent d'augmenter l'efficacité du programme généré, telle que la possibilité d'insérer des lignes assembleur directement dans le code C, d'appeler des fonctions assembleur à partir du code C, et l'utilisation des opérateurs intrinsics*.

*: seult avec les DSP Texas?

La programmation en C est intéressante pour réduire le temps de développement d'une application, et on l'utilise pour les parties du programme dont le temps d'exécution n'est pas critique. On utilise la programmation en assembleur pour les blocs de programme effectuant des calculs arithmétiques intensifs dont le temps d'exécution est critique.

Les convertisseurs CAN fournissent une représentation numérique de chaque échantillon. Cette représentation peut ensuite être transformée par le DSP selon le type d'arithmétique utilisée.

On se limite ici à la quantification scalaire, c'est-à-dire à la quantification d'un échantillon isolé. On distingue plusieurs types de quantification scalaire :

- la quantification uniforme;
- la quantification non uniforme, en particulier la conversion de type logarithmique.

La *figure 2.2* fixe les notations utilisées pour le quantificateur.

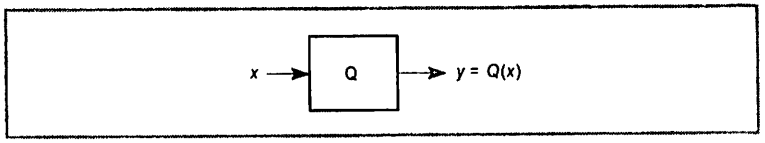

Figure 2.2 – Schéma d'un quantificateur.

Quantifier une grandeur x pouvant prendre une valeur quelconque dans un intervalle $[-x_{max}, x_{max}]$ consiste à remplacer x par une valeur quantifiée $Q(x) = y_i$, choisie parmi un ensemble fini (ou dénombrable) de N valeurs possibles.

On appelle les valeurs y_i les valeurs de quantification. Le choix de la valeur de quantification pour un x donné est déterminé en fonction de $N + 1$ valeurs de décision x_i, par la règle suivante :

$$x \in [x_i, x_{i+1}[\Rightarrow Q(x) = y_i$$

La *figure 2.3* est un exemple de répartition des valeurs de quantification et de décision.

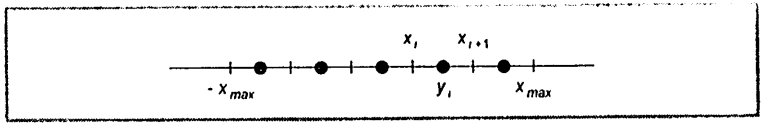

Figure 2.3 – Exemple de valeur de quantification.

2.1.1 Quantification uniforme

Lorsque la largeur des intervalles de décision est constante, on parle de quantification uniforme ou linéaire. La largeur des intervalles est appelée pas de quantification; elle est notée q.

$$x_{i+1} - x_i = q = \text{pas de quantification constant}$$

Le pas de quantification q peut s'exprimer en fonction des valeurs extrêmes par :

$$q = \frac{2x_{max}}{2^N}$$

On définit le facteur de surcharge, noté Γ, comme le rapport entre la valeur maximale du convertisseur x_{max} et l'écart type des échantillons à convertir, noté σ_x.

$$\Gamma = \frac{x_{max}}{\sigma_x}$$

Lors de la quantification, deux types d'erreur peuvent être commis :

- l'erreur de granulation;
- l'erreur de saturation.

Une erreur de saturation se produit lorsque l'amplitude de l'échantillon à convertir est supérieure à x_{max}. Cette erreur est d'autant plus gênante qu'elle n'est pas bornée, on cherche donc à minimiser la probabilité de saturation. La probabilité de saturation p_D dépend de la valeur de Γ. Pour des échantillons gaussiens :

$$\Gamma = 2 \Rightarrow p_D = 0{,}045$$
$$\Gamma = 4 \Rightarrow p_D = 0{,}000\ 06$$

Une erreur de granulation se produit sur les échantillons d'amplitude inférieure à x_{max} en valeur absolue. C'est la différence entre l'échantillon et sa valeur quantifiée. Cette erreur est bornée. Si la quantification s'effectue par arrondi au plus proche voisin, l'erreur de granulation e_g en valeur absolue est inférieure à $q/2$.

$$e_g = x - Q(x)$$

$$|e_g| \leq \frac{q}{2}$$

Sous certaines hypothèses relativement générales, on peut calculer la valeur moyenne et l'écart type de cette erreur :

$$E(e_g) = 0$$

$$E(e_g^2) = \sigma_g^2 = \frac{q^2}{12}$$

Avec les mêmes hypothèses, le rapport signal sur bruit (noté RSB_{dB}) entre la puissance du signal σ_x^2 et la puissance de l'erreur de granulation σ_g^2 peut s'exprimer en décibels par la relation suivante (où N représente le nombre de bits du convertisseur) :

$$RSB_{dB} = 10\log_{10}\left(\frac{\sigma_x^2}{\sigma_g^2}\right)$$

$$RSB_{dB} \approx 10\log_{10}\left(\sigma_x^2\right) + 6N - 10\log_{10}\left(x_{max}^2\right) + 10\log_{10}\left(\frac{3}{2}\right)$$

$$RSB_{dB} \approx 6N + 10\log_{10}\left(\frac{3}{2}\right) - 20\log_{10}(\Gamma)$$

Le rapport signal sur bruit en décibels dépend donc de façon linéaire du nombre de bits de quantification et de la puissance du signal en décibels.

2.1.2 Quantification logarithmique

La quantification de type logarithmique permet d'obtenir un rapport signal sur bruit de quantification à peu près constant, quelle que soit la puissance du signal.

L'écart entre les seuils de décision n'est pas constant. Il croît de façon logarithmique en fonction de l'amplitude du signal à quantifier. Une quantification logarithmique peut se réaliser par une compression des amplitudes suivie d'une quantification uniforme, puis d'une expansion des amplitudes.

Lois de compression expansion

La *figure 2.4* représente un convertisseur logarithmique composé d'une quantification uniforme précédée d'une compression et suivie d'une expansion.

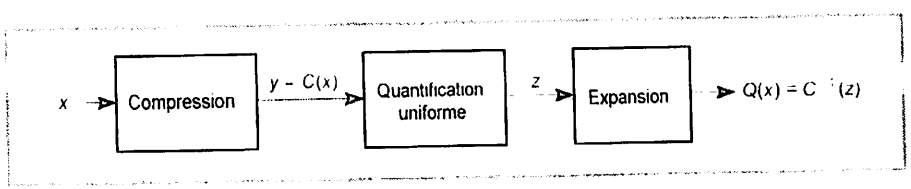

Figure 2.4 – Conversion logarithmique.

La loi de compression est notée *C(x)*. La loi d'expansion est l'opération inverse.

$$-x_{max} \le x \le x_{max} \qquad y = C(x) \quad x = C^{-1}(y)$$

La loi de compression doit approcher d'une fonction logarithme. Deux lois sont très utilisées en pratique : la loi A et la loi μ. Ces deux lois sont appliquées dans les codecs : circuits de conversion analogiques/numériques utilisés en téléphonie filaire. La loi A est appliquée en Europe, la loi μ aux États-Unis et au Japon. Ces 2 lois sont très proches.

Description de la loi A

La définition de la fonction de compression *C(x)* fait intervenir une constante appelée A.

$$C(x) = \frac{A|x|}{1+\log(A)}\,sign(x) \qquad 0 \le \frac{|x|}{x_{max}} < \frac{1}{A}$$

$$C(x) = x_{max}\,\frac{1+\log\left(A|x|/x_{max}\right)}{1+\log(A)}\,\mathrm{sign}(x) \qquad \frac{1}{A} \le \frac{|x|}{x_{max}} \le 1$$

$A = 87{,}56$

La *figure 2.5* représente la loi de compression A. Sur la figure, on a normalisé x_{max} à 1.

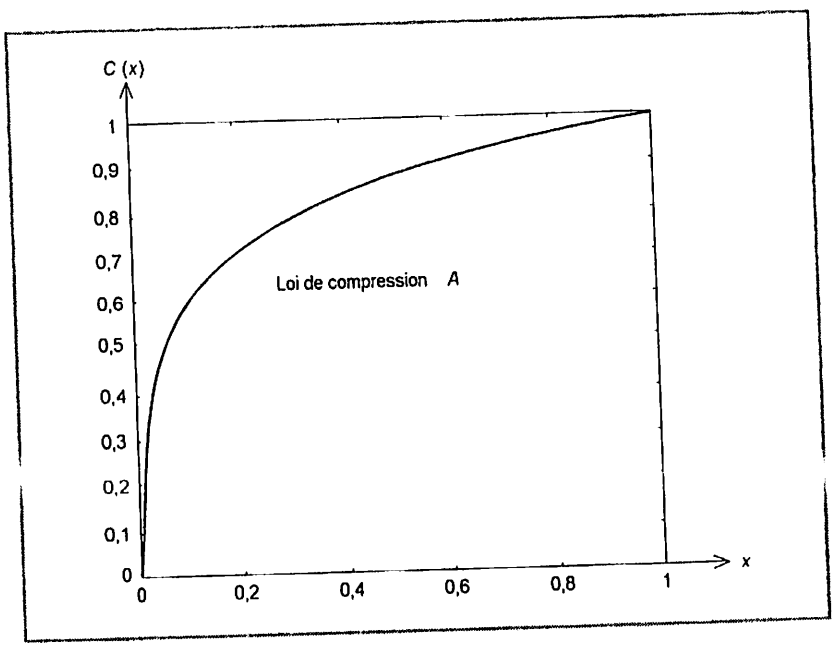

Figure 2.5 – Loi de compression A.

- *Approximations par segments des lois de compression* A *et* μ

  Pour leur réalisation matérielle, les lois A et μ sont approchées par des segments de droite. La loi A est approchée par une courbe à 13 segments, et la loi μ par une courbe à 15 segments. Elles sont appliquées dans ce cas-là avec une numérisation sur 8 bits.

En ce qui concerne la loi A, la pente du premier segment passant par l'origine est de 16. Puis, les pentes des segments successifs sont obtenues par division par deux. La pente du dernier segment vaut donc 1/4.

La *figure 2.6* représente la loi A à 13 segments; x_{max} est encore normalisé à 1.

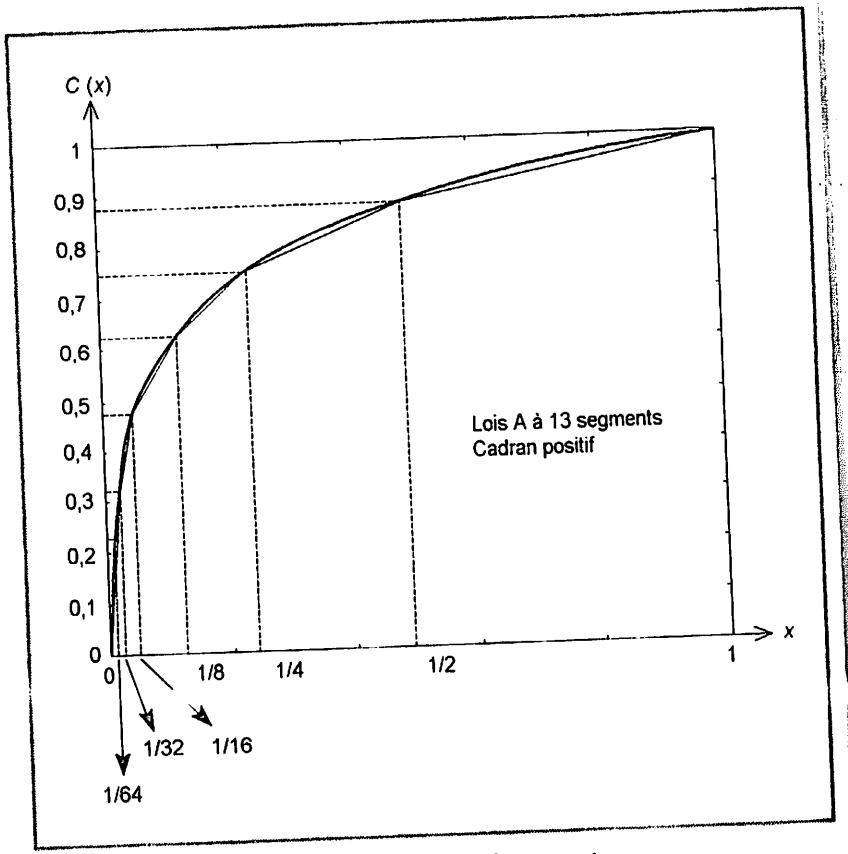

Figure 2.6 – Loi A à 13 segments.

Le rapport signal sur bruit de quantification obtenu avec la loi A est constant sur une large plage de puissance de signal. Dans le cas de la conversion sur 8 bits, on peut remarquer que les petits signaux sont amplifiés par un facteur 16 avant d'être convertis, ce qui revient à diviser par 16 le pas de quantification, c'est-à-dire à utiliser 12 bits de quantification (gain de 4 bits). Par contre, pour les grands signaux, le pas de quantification est multiplié par 4 par rapport à un convertisseur 8 bits uniforme et on perd donc 2 bits.

Le rapport signal sur bruit de quantification en décibels pour la loi A s'écrit :

$$x \text{ grand} : RSB_{dB} \approx 6N + 4{,}77 - 20\log(1 + \ln A)$$

$$x \text{ petit} : RSB_{dB} \approx 6N + 4{,}77 + 10\log\left(\frac{A}{1 + \ln A}\right) - 20\log(\Gamma)$$

$$\Gamma = \frac{x_{max}}{\sigma_x}$$

N représente le nombre de bits utilisés et Γ le facteur de charge.

Pour le signal téléphonique, la qualité subjective obtenue avec une conversion selon la loi A sur 8 bits est équivalente à celle obtenue avec une conversion uniforme sur 12 bits.

Le rapport signal sur bruit maximal est meilleur pour la conversion uniforme sur 12 bits puisqu'il est de l'ordre de 70 dB au lieu de 38 dB pour la conversion logarithmique sur 8 bits. Mais dans le cas de la conversion logarithmique sur 8 bits, la dynamique de signal pour laquelle le rapport signal sur bruit maximal est obtenu est grande (une trentaine de décibels), alors que pour la conversion uniforme, le RSB (en dB) décroît proportionnellement avec la puissance du signal en décibels.

2.2 Représentation binaire des nombres et arithmétique en précision finie

Les processeurs de traitement de signal peuvent travailler sur des données représentées en virgule fixe ou en virgule flottante. Pour certains algorithmes, il peut être intéressant de les faire fonctionner en virgule flottante par bloc.

Avant de préciser ces différentes représentations, il est utile de rappeler les représentations binaires les plus courantes des entiers relatifs.

2.2.1 Représentation binaire des entiers relatifs

Plusieurs représentations binaires des entiers relatifs sont couramment utilisées, en particulier dans les convertisseurs analogiques/numériques. On peut citer :

- complément à 2;
- complément à 1;
- signe, valeur absolue;
- binaire décalé.

Les *tableaux 2.1* et *2.2* illustrent ces diverses représentations avec un exemple sur 3 bits.

Tableau 2.1 – Nombres non signés codés en binaire pur.

Entiers positifs	Binaire pur
7	111
6	110
5	101
4	100
3	011
2	010
1	001
0	000

Tableau 2.2 – Différentes représentations binaires.

Entiers relatifs	Binaire décalé	Signe plus valeur absolue	Complément à 1	Complément à 2
+ 3	111	011	011	011
+ 2	110	010	010	010
+ 1	101	001	001	001
0		000	000	
0	100	100	111	000
– 1	011	101	110	111
– 2	010	110	101	110
– 3	001	111	100	101
– 4	000			100

Les représentations binaires en complément à 1, et en complément à 2 diffèrent par la représentation des nombres entiers négatifs.

Dans la représentation en complément à 1, un entier négatif x est codé par la représentation binaire pure de l'entier positif y égal au complément à 1 de x :

$$y = 2^N - |x| - 1$$

Le terme complément à 1 représente en fait le complément à $2^N - 1$; entier positif qui sur N bits s'écrit avec tous les bits à 1.

Dans la représentation en complément à 2, un entier négatif x est codé par la représentation binaire pure de l'entier positif y égal au complément à 2 de x :

$$y = 2^N - |x|$$

Le terme complément à 2 représente en fait le complément à 2^N.

2.2.2 Entiers relatifs en complément à 2

La représentation des nombres entiers relatifs utilisée dans les DSP comme dans la plupart des microprocesseurs est la représentation binaire en complément à deux.

Soit un entier relatif x, sa représentation binaire en complément à 2 sur N bits est constituée de la suite de bits b_i :

$$x \rightarrow b_{N-1} b_{N-2} \cdots b_k \cdots b_1 b_0$$

La relation entre x et les valeurs des bits b_i est la suivante :

- pour x positif, la représentation en complément à deux correspond à la représentation binaire pure, c'est-à-dire à une représentation pondérée en base 2 :

$$x \geq 0 \Rightarrow x = \sum_{i=0}^{N-1} b_i 2^i \quad \text{et} \quad b_{N-1} = 0$$

- pour x négatif, la représentation en complément à 2 de x est la représentation binaire pure du complément à 2^N de $x(y)$:

$$x < 0 \Rightarrow y = 2^N - |x| \Rightarrow y = \sum_{i=0}^{N-1} b_i 2^i$$

Dans tous les cas, on peut écrire la relation suivante :

$$x = -2^{N-1} b_{N-1} + \sum_{i=0}^{N-2} b_i 2^i$$

Le bit de poids le plus fort vaut 0 pour les entiers positifs et 1 pour les entiers négatifs.

Propriétés de la représentation en complément à 2

Valeurs extrêmes

Les valeurs extrêmes représentables en complément à 2 sur N bits sont :

$$\max = 2^{N-1} - 1$$

$$\min = -2^{N-1}$$

Mode d'overflow

La représentation en complément à 2 est une représentation circulaire. Lorsque l'on ajoute 1 à la plus grande valeur positive, on obtient la valeur négative de plus grande valeur absolue. Si on enlève 1 à la valeur négative de plus grande valeur absolue, on obtient la plus grande valeur positive.

$$(2^{N-1}-1)+1 = 2^{N-1} \Leftrightarrow -2^{N-1}$$

Cette particularité peut avoir des conséquences fâcheuses. Ainsi lorsque le gain d'un filtre numérique est trop grand pour que la sortie puisse s'exprimer sur N bits, au lieu de saturer comme en analogique, la sortie oscille entre des valeurs de grandes amplitudes positives et négatives. On parle de cycles limites de grande amplitude. Les débordements, en complément à 2, génèrent ainsi des pics difficiles à filtrer.

Les DSP disposent en général d'une arithmétique de saturation. Ils peuvent être configurés soit pour travailler en vrai complément à 2, soit pour travailler en complément à 2 avec saturation arithmétique. Dans ce dernier cas, si le résultat d'un calcul est en valeur absolue supérieur à la plus grande valeur représentable en complément à 2 dans l'accumulateur, l'arithmétique de saturation détecte le débordement et le processeur remplace ce résultat par une valeur d'écrêtage : la plus grande valeur positive ou négative représentable, c'est-à-dire qu'il réalise une saturation. Dans les DSP de la famille '54x, le mode de fonctionnement est déterminé par la valeur du bit de mode OVM (*Overflow Mode*).

La *figure 2.7* représente une sinusoïde d'amplitude supérieure à ce qui peut être représenté avec les N bits. On suppose que la plus grande amplitude représentable est 0,75 alors que l'amplitude de la sinusoïde est 1. La sinusoïde est en trait plein, la sinusoïde écrêtée par une arithmétique de saturation est représentée en tirets longs-tirets courts et la sinusoïde en complément à 2 avec *overflow* est en pointillés.

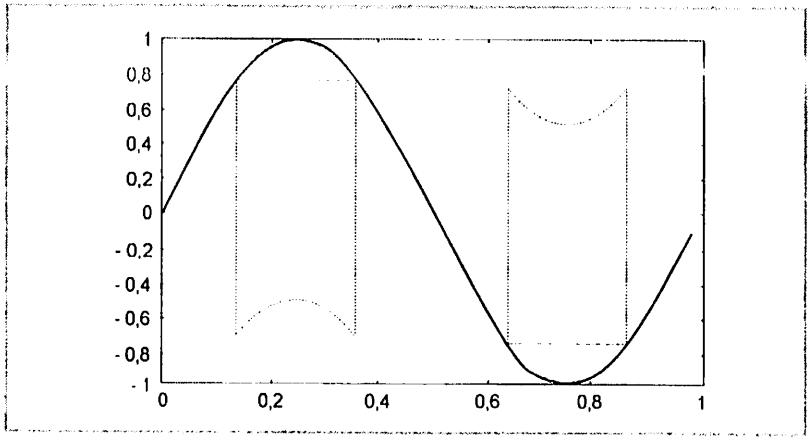

Figure 2.7 – Saturation et écrêtage lors d'un débordement.

Extension du bit de signe

Lorsque l'on connaît la représentation en complément à 2 sur N bits d'un nombre x, pour obtenir la représentation en complément à 2 du même nombre sur M bits ($M > N$), il suffit d'étendre le bit de signe, c'est-à-dire de conserver les N bits de poids faibles et de remplir les bits de poids forts supplémentaires en répétant la valeur du bit de signe (le MSB sur N bits).

Lorsque l'on charge une donnée 16 bits dans un accumulateur 32 bits, le bit de signe est automatiquement étendu. Dans certains cas, cette extension du bit de signe n'est pas souhaitable, par exemple pour une donnée 16 bits représentant une adresse, grandeur forcément positive. Il est en général possible de configurer les DSP pour qu'ils effectuent ou non une extension du bit de signe lors d'un chargement dans l'accumulateur. Pour les '54x, le bit de mode SXM (*Sign eXtension Mode*) permet de choisir le mode désiré.

Addition/soustraction en complément à 2

En complément à 2, les additions et les soustractions sont simples à réaliser : pour ajouter 2 nombres entiers signés N bits avec un résultat sur N bits, quel que soit le signe des nombres, il suffit d'ajouter les codes en complément à deux.

La *figure 2.8* donne quelques exemples d'addition avec $N = 3$ bits. On y a mis en évidence la circularité de cette représentation, ainsi que le bit de retenue ou bit de *carry*.

Lorsque le résultat d'une addition est supérieur au plus grand nombre représentable, il y a débordement ou *overflow*. Dans les '54x, lorsqu'il y a *overflow*, le bit d'état OV passe à 1.

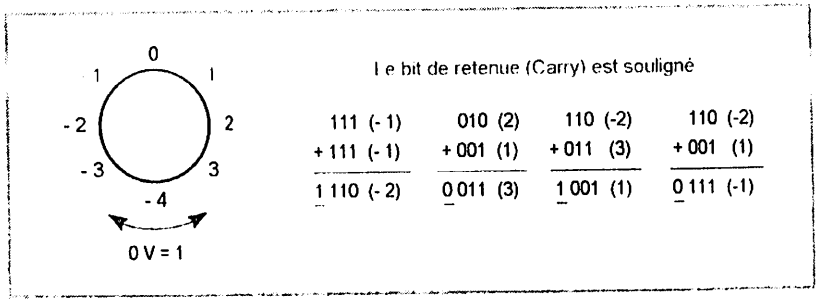

Figure 2.8 – Représentation circulaire et bit de retenue.

De plus, si le résultat d'une série d'additions et/ou de soustractions peut s'exprimer sur les N bits de l'accumulateur, le résultat est correct même si des débordements intermédiaires se produisent.

EXEMPLE SUR 3 BITS

Soit à calculer 3 + 1 – 2, le résultat théorique vaut 2, ce qui peut s'exprimer sur 3 bits.

Si on ajoute les données deux par deux, on effectue successivement les opérations (3 + 1) dont le résultat est noté R, puis (R – 2).

Le résultat de 3 + 1 vaut 4 en théorie, mais sur 3 bits on obtient : R = 011 + 001 = 100, c'est-à-dire R = – 4 car il y a débordement. Ce résultat intermédiaire est faux, mais lors de l'opération suivante R – 2 on obtient 100 – 010 = 010, c'est-à-dire 2, ce qui est bien le résultat correct.

Pour exprimer sans risque de débordement le résultat de l'addition de 2 nombres N bits, il faut au moins N + 1 bits.

La simplicité de l'addition est la raison principale pour laquelle la représentation en complément à 2 est utilisée dans la majorité des processeurs numériques.

Multiplication et décalage en complément à 2

Le résultat de la multiplication de 2 nombres de N bits s'écrit sur $2N$ – 1 bits.

En général, le registre produit est sur $2N$ bits et de ce fait, les deux bits de poids forts du résultat d'un produit sont identiques (extension du bit de signe quand on passe de $2N$ – 1 bits à $2N$ bits). Le MSB est ici inutile. Aussi est-il possible dans la famille '54x de décaler d'un bit à gauche le résultat des produits de façon systématique, sans temps de cycle supplémentaire ni instruction particulière : il suffit de configurer à 1 le bit FRCT.

Dans un DSP travaillant en complément à 2, la multiplication est câblée, par contre les divisions doivent se faire par logiciel.

Les multiplications ou divisions par une puissance de 2 peuvent être effectuées par des décalages arithmétiques à gauche ou à droite respectivement. Ces décalages de quelques bits peuvent en général s'effectuer en un seul temps de cycle. Lors des décalages arithmétiques à droite, le bit de signe est étendu. Lors des décalages à gauche, des zéros sont introduits dans les bits de poids faibles.

2.2.3 Représentation binaires des nombres réels en précision finie

Deux approches sont utilisées pour la représentation binaire des nombres réels en précision finie :

– la représentation en virgule fixe ou format fixe ;

– la représentation en virgule flottante ou format flottant.

Les DSP sont conçus pour l'une ou l'autre de ces représentations. Toutefois, un DSP travaillant en virgule fixe pourra aussi effectuer des calculs en virgule flottante, mais de manière peu efficace et réciproquement.

Représentation binaire des nombres fractionnaires en format virgule fixe

On appelle représentation en virgule fixe des nombres fractionnaires, ou plus généralement des nombres réels avec une précision finie, une représentation comprenant une partie entière suivie d'une partie fractionnaire correspondant à des bits après la virgule.

On utilise souvent l'expression « format Q_k » pour indiquer une représentation comportant k bits derrière la virgule.

Soit un nombre x réel quelconque, sa représentation binaire en virgule fixe, en précision finie sur N bits en format Q_k (c'est-à-dire avec k bits derrière la virgule) s'écrira :

$$x \rightarrow \overbrace{b_{N-1-k}\cdots b_1 b_0}^{\textit{Partie entière}}, \overbrace{b_{-1}b_{-2}\cdots b_{-k}}^{\textit{Partie fractionnaire}}$$

Elle correspond à la représentation du nombre entier y obtenu en arrondissant à l'entier le plus proche le nombre réel formé du produit de x par 2^k.

$$y = \lfloor 2^k x \rceil$$

[...] signifie arrondi au plus proche voisin.

Par la suite on suppose que cette représentation de y est faite en complément à 2.

Ainsi, la représentation binaire :

$$\overbrace{b_{N-1-k}\cdots b_1 b_0}^{\textit{Partie entière}}, \overbrace{b_{-1}b_{-2}\cdots b_{-k}}^{\textit{Partie fractionnaire}}$$

correspond au nombre fractionnaire :

$$z = -b_{N-1-k}2^{N-1-k} + b_{N-2-k}2^{N-2-k} + \cdots b_0 + b_{-1}2^{-1} + \cdots + b_{-k}2^{-k}$$

Ce nombre est une approximation en précision finie du réel x, sur N bits avec k bits derrière la virgule.

Un nombre réel étant rarement de précision finie, la représentation sur un nombre fini de bits introduit une erreur. En virgule fixe sur N bits avec format Q_k, cette erreur est inférieure à 2^{-k}, si $N - k$ bits suffisent pour la partie entière.

Valeurs extrêmes en virgule fixe sur N bits avec format Q_k

Les valeurs extrêmes représentables sont :

$$max = 2^{N-1-k} - 2^{-k}$$

$$min = -2^{N-1-k}$$

Dynamique et précision en virgule fixe sur N bits avec format Q_k

En virgule fixe sur N bits avec k bits de partie fractionnaire, il est possible de représenter les réels compris entre -2^{N-1-k} et $2^{N-1-k}-2^{-k}$ avec une erreur inférieure à 2^{-k} en valeur absolue.

Pour les réels en dehors de cette plage, on dit qu'il y a :

– *overflow* si le nombre est trop grand en valeur absolue. Il y a débordement ;

– *underflow* si le nombre est trop petit. Il est alors représenté par zéro.

En conclusion, l'erreur absolue est inférieure à 2^{-k} sur une plage de valeurs correspondant à une dynamique de $6N$ dB. La dynamique est ici définie comme 2 fois le rapport entre la plus petite grande et la plus petite des valeurs positive exprimables.

Exemple de représentation en virgule fixe sur N = 8 bits en format Q_5

Le terme format Q_5 signifie qu'il y a 5 bits derrière la virgule.

On travaille ici en complément à deux.

La partie entière est formée de 3 bits (8 – 5). Elle permet de représenter des entiers relatifs compris entre 3 et – 4.

La partie fractionnaire, sur 5 bits, permet de représenter des nombres compris entre 0 et 0,968 75 ce qui correspond à la somme des 5 premières puissances de 2 négatives.

Le *tableau 2.3* donne quelques représentations et leurs équivalences décimales.

Tableau 2.3 – Représentations binaires et valeurs décimales.

Représentation binaire virgule fixe, format Q_5	Valeur décimale
011 10000	3,5
001 10100	1,625
110 10001	– 1,468 75
100 00000	– 4
011 11111	3,968 75

En format Q_5 sur 8 bits, les valeurs extrêmes représentables sont – 4 et 3,968 75.

La représentation sur un nombre fini de bits introduit une erreur. Cette erreur est inférieure à 2^{-k}, soit 1/32 dans l'exemple.

Le *tableau 2.4* donne quelques exemples de nombres réels, leurs représentations sur 8 bits en format Q_5 avec l'équivalence décimale et l'erreur commise par cette représentation.

Tableau 2.4 – Représentations binaires et valeurs décimales.

Valeur réelle	Représentation binaire virgule fixe, format Q_5	Équivalence décimale	Erreur commise
1/3	000 01011	0,343 75	0,010 416 666...
$\sqrt{2}$	001 01101	1,406 25	0,007 963 562...
π	011 00101	3,156 25	0,014 657 346...

Addition de nombres fractionnaires en virgule fixe

Lors de l'addition de nombres fractionnaires en virgule fixe, il faut comme en décimal aligner les virgules. La somme de 2 nombres en format Q_k donne un résultat en format Q_k :

$$Q_k + Q_k \Rightarrow Q_k$$

Multiplication de 2 nombres fractionnaires en virgule fixe

Le produit de nombres en virgule fixe sur N bits donne un résultat sur $2N-1$ bits.

Comme en décimal, le nombre de bits après la virgule du résultat est égal à la somme des nombres de bits derrière la virgule des 2 opérandes :

$$Q_L \cdot Q_F \Rightarrow Q_{L+F}$$

Représentation binaire des nombres fractionnaires en virgule flottante

Dans la représentation binaire en virgule flottante en précision finie sur N bits, les nombres sont représentés par une mantisse M et un exposant E; M et E représentent la valeur x :

$$x = M \cdot 2^E$$

Pour une représentation sur N bits, la mantisse M est exprimée sur m bits, et l'exposant E sur e bits, avec $N = m + e$.

Pour que la représentation soit unique, M est normalisée : $\frac{1}{2} \leq |M| < 1$ par exemple.

Plage de nombres représentables

Sur N bits, avec m bits pour la mantisse, e bits pour l'exposant et une mantisse normalisée entre 0,5 et 1, on peut représenter des nombres dont la valeur absolue est comprise dans l'intervalle :

$$\left[\frac{1}{2}2^{-(2^e-1)}, \left(1-2^{1-m}\right)2^{2^e-1}\right]$$

Overflow si $|x| > \left(1-2^{1-m}\right)2^{(2^e-1)}$

Underflow si $|x| < \frac{1}{2}2^{-(2^e-1)}$

Interprétation de la représentation binaire virgule flottante

Dans une représentation en virgule flottante, les nombres sont répartis sur une échelle non linéaire, comme indiquée sur la *figure 2.9* où l'on n'a représenté que les nombres positifs.

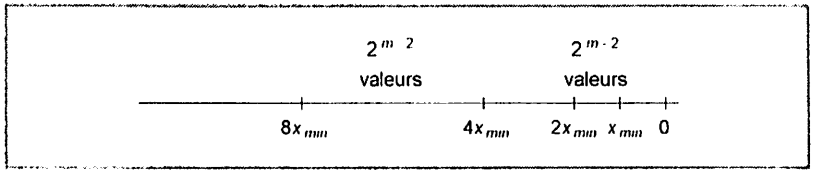

Figure 2.9 – Répartition sur une échelle non linéaire.

Ainsi pour les nombres positifs, la plage de valeurs représentables comprises entre :

$$x_{min} = \frac{1}{2}2^{-2^e-1} \text{ et } x_{max} = \left(1-2^{1-m}\right)2^{2^e-1}$$

est partagée en intervalles de largeur en progression géométrique de raison égale à 2. Chaque intervalle correspond à une valeur de l'exposant E et contient 2^{m-2} valeurs associées aux différentes valeurs positives possibles de la mantisse entre 0,5 et 1 sur m bits. La largeur du premier intervalle est égale à x_{min}.

Lorsque x vaut 0, son exposant est égal à 2^{e-1}.

On peut remarquer que la précision absolue de la représentation est meilleure pour les valeurs de faible amplitude que pour les valeurs de forte amplitude.

Dans le premier intervalle $[x_{min}, 2x_{min}]$, l'erreur de représentation est inférieure à :

$$\frac{1}{2}2^{-2^e-1}2^{2-m}$$

Dans le dernier intervalle $[0{,}5x_{max}, x_{max}]$, l'erreur de représentation est inférieure à :

$$\left(1-2^{1-m}\right)2^{2^e-1-1}2^{2-m}$$

La précision relative est à peu près constante.

Pour un nombre de bits donné, le nombre de bits de la mantisse détermine la précision et le nombre de bits de l'exposant détermine la dynamique.

La dynamique D, ou plus précisément le rapport entre la plus grande et la plus petite des valeurs positives exprimables, exprimée en dB, vaut :

$$D = 20\log_{10}\left(2\left(1-2^{1-m}\right)2^{2^e}\right) \approx 6\left(2^e+1\right)\text{dB}$$

Exemple de représentation en virgule flottante sur 8 bits avec m = 5 et e = 3

On suppose que la mantisse et l'exposant sont exprimés en complément à 2. La mantisse M est normalisée :

$$\frac{1}{2} \le |M| < 1$$

On suppose de plus que la mantisse est écrite avant l'exposant : ME.

Le *tableau 2.5* donne quelques représentations et leurs équivalences décimales.

Tableau 2.5 – Représentations binaires et équivalences décimales.

Représentation binaire Mantisse M	Exposant E	Valeur décimale $M2^E$
01110	010	1,75
01100	100	0,046 875
10010	011	– 7
01000	100	0,031 25
01111	011	7,5

Pour cette représentation, les valeurs positives extrêmes représentables sont 0,031 25 et 7,5.

Tableau 2.6 – Représentations binaires et équivalences décimales.

Valeur réelle	M	E	Équivalence décimale	Erreur commise
1/3	01011 (0,687 5)	111 (– 1)	$0{,}687\,5 \times 2^{-1} = 0{,}343\,75$	0,013 416 6… 4 %
$\sqrt{2}$	01011 (0,687 5)	001 (1)	$0{,}687\,5 \times 2^1 = 1{,}375$	– 0,039 213 5… 2,77 %
π	01101 (0,812 5)	010 (2)	$0{,}812\,5 \times 2^2 = 3{,}25$	0,108 407 34… 3,4 %
$\sqrt{50}$	01110 (0,875)	011 (3)	$0{,}875 \times 2^3 = 7$	0,071 067 81… 1 %

Par ailleurs, la représentation sur un nombre fini de bits introduit une erreur. Le *tableau 2.6* donne quelques exemples de nombres réels, leurs représentations sur 8 bits en virgule flottante (m = 5, e = 3) avec l'équivalence décimale, et l'erreur commise par cette représentation.

Addition en format virgule flottante

Pour ajouter 2 nombres A et B en virgule flottante, il faut dénormaliser le nombre le plus petit (B) :

$$A + B = M_A 2^{E_A} + M_B 2^{E_B} = \left(M_A + M_B 2^{E_B - E_A}\right) 2^{E_A}$$

Cette dénormalisation complique les opérations par rapport à une addition en virgule fixe; elle fait perdre de la précision sur la représentation du plus petit nombre à cause de l'arrondi de la mantisse.

Multiplication en virgule flottante

Soient 2 nombres A et B en virgule flottante, on peut écrire :

$$A \cdot B = M_a M_b 2^{E_a + E_b} = M 2^E$$

Pour obtenir M et E, il faut normaliser le produit des mantisses $M_a M_b$ et corriger l'exposant : il faut $2m - 1$ bits pour exprimer exactement M et $e + 1$ bits pour exprimer E.

Si on tronque M à m bits, l'erreur absolue augmente vite.

Les processeurs virgule flottante effectuent effectivement ces opérations.

Comparaison virgule fixe, virgule flottante

La numérisation virgule fixe sur N bits correspond à une échelle linéaire et à une quantification uniforme.

La numérisation virgule flottante sur N bits correspond à une échelle non linéaire en progression géométrique de raison 2 et à une quantification non uniforme de type quasi logarithmique.

- *Erreur de quantification en virgule fixe*

  Soit x une valeur réelle et $\hat{x}$ sa représentation en virgule fixe sur N bits en format Q_k.

  On suppose que x est inférieur à x_{max}, la plus grande valeur représentable.

  Sous certaines hypothèses assez générales sur la distribution de x, on peut considérer que l'erreur d de représentation est une variable uniforme et on peut écrire :

$$d = \hat{x} - x \quad (arrondi)$$

$$|d| \le \frac{q}{2} \qquad \text{pour} |x| \le x_{max}$$

$$q = 2^{-k}$$

$$x_{max} = 2^{N-1-k} - 2^{-k} = q\left(2^{N-1} - 1\right)$$

$$E(d) = 0$$

$$E\left(d^2\right) = \sigma_d^2 = \frac{q^2}{12}$$

$$RSB_{dB} = 10 \log_{10}\left(\frac{\sigma_x^2}{\sigma_d^2}\right)$$

$$RSB_{dB} \approx 10 \log_{10}\left(\sigma_x^2\right) + 6N - 10 \log_{10}\left(x_{max}^2\right) + 10 \log_{10}\left(\frac{3}{2}\right)$$

- *Erreur de quantification en virgule flottante*

  Soit x une valeur réelle et $\hat{x}$ sa représentation en virgule flottante sur N bits avec m bits de mantisse et e bits d'exposant.

  On suppose que x est inférieur à x_{max}, la plus grande valeur représentable.

  On peut écrire :

$$d = \hat{x} - x \quad (arrondi)$$

$$d_m = \text{erreur d'arrondi sur la mantisse}$$

$$0 \le |d_m| \le \frac{1}{2} 2^{-(m-1)}$$

$$d_r = \text{erreur relative sur } x = \frac{d}{x} = \frac{d_m}{M}$$

$$|d_r| \le 2^{-(m-1)}$$

Sous certaines hypothèses assez générales sur la distribution de x, on peut montrer que :

$$d_r \text{ est un bruit blanc non corrélé avec } x$$

$$d = \hat{x} - x = x d_r$$

$$\sigma_d^2 = \sigma_x^2 \sigma_{d_r}^2$$

$$RSB_{dB} = 6m + 1{,}44$$

Ici le RSB ne dépend pas de la puissance du signal, à la différence de la représentation en virgule fixe.

La *figure 2.10* illustre le rapport signal sur bruit obtenu avec N = 16 bits en virgule fixe, et en virgule flottante pour m = 12 et e = 4.

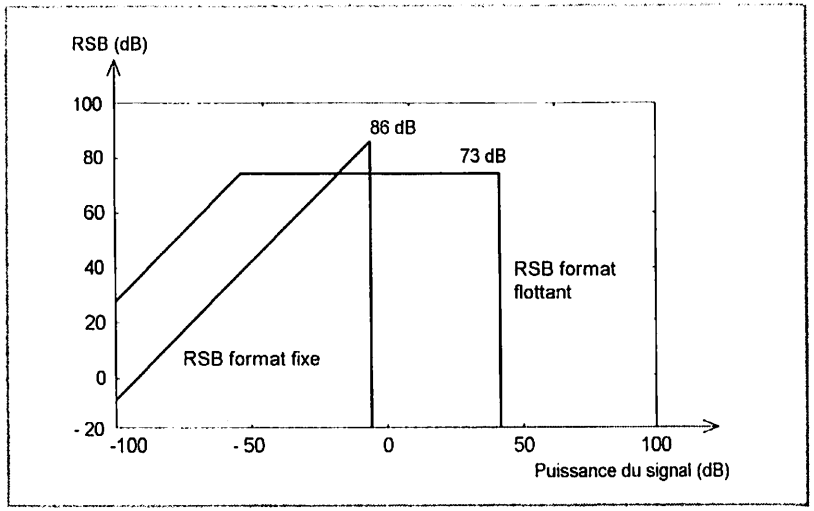

Figure 2.10 – Rapport signal à bruit en format fixe et flottant.

Comparaison des dynamiques

On appelle dynamique, le rapport en décibels entre la plus grande et la plus petite amplitude non nulle représentables.

Le *tableau 2.7* effectue une comparaison des dynamiques obtenues avec les formats fixes et flottants.

Tableau 2.7 – Comparaison des dynamiques en format fixe et flottant.

	Virgule fixe *N* bits Format Q_k	Virgule flottante *N* bits *M* sur *m* bits et *E* sur *e* bits
Dynamique	6N dB	$20\log_{10}(2(1-2^{1-m})2^{2^e}) \approx 6(2^e+1)$ dB

Le *tableau 2.8* compare la dynamique et la précision obtenues en virgule fixe et virgule flottante pour N = 32 bits.

Tableau 2.8 – Dynamique et précision pour N = 32 bits.

	Virgule fixe N = 32	Virgule flottante N = 32, m = 24, e = 8
Dynamique	$> 10^9$	$> 10^{77}$
Précision	Précision max > 9 digits	Précision toujours > 7 digits

Conclusion

Pour un nombre de bits *N* donné, la représentation des nombres en virgule flottante réalise un compromis entre dynamique (*E*) et précision (*M*).

En virgule fixe, les opérateurs de traitement sont simples mais il faut surveiller le cadrage des données pour éviter les débordements, tout en conservant un maximum de précision.

En virgule flottante, les opérateurs sont plus complexes mais on dispose d'une plus grande dynamique pour une précision minimale donnée et le cadrage des données est moins critique.

Format IEEE 754, virgule flottante

Le format IEEE 754 de représentation des nombres en virgule flottante possède les principales caractéristiques suivantes :

Pour N = 32 bits :

- un bit de signe *S*;
- un exposant sur 8 bits;
- une fraction sur 23 bits.

L'exposant est représenté en binaire décalé avec un biais égal à 127.

La mantisse constituée du bit de signe et de la fraction est exprimée en signe plus valeur absolue. La valeur absolue est normalisée en binaire entre 1.00...00 et 1.11...11, et comme le premier bit vaut toujours 1, il est caché (non représenté) et on stocke seulement la partie fractionnaire.

EXEMPLE POUR N = 32 BITS :

Le nombre x = 28 est représenté de la façon suivante :

$x = 28 = 1{,}75 \times 2^4 \rightarrow$ 0 10000011 1100.0.

C'est-à-dire :

- S = 0 : nombre positif;
- E = 4 est représenté en binaire décalé avec un biais de 127. Il est donc exprimé par la représentation binaire pure de 127 + 4 = 131, soit en binaire 10000011;
- M = 1,75 : la partie fractionnaire de la mantisse vaut 0,75. La représentation binaire correspondante est 1100.0, compte tenu du bit caché.

Le format IEEE 754 définit également :

- la double précision étendue sur 64 bits;
- la simple précision étendue sur 43 bits;
- la double précision étendue sur 79 bits.

Les caractéristiques correspondantes sont résumées dans le *tableau 2.9*.

Tableau 2.9 – Caractéristiques des formats IEEE flottants.

Nombre de bits	Signe	Exposant	Fraction	Total
Double précision 64 bits	1	11	52	64
Simple précision étendue 43 bits	1	11	31	43
Double précision étendue 79 bits	1	15	63	79

Les DSP ne respectent pas forcément le format IEEE 754 de représentation des nombres en virgule flottante.

Virgule flottante par bloc

Dans certains cas, en particulier lorsqu'on utilise un DSP virgule fixe pour effectuer des calculs nécessitants à la fois une grande précision et une grande dynamique (une FFT par exemple), il peut être intéressant de travailler en virgule flottante par bloc.

Dans la représentation en virgule flottante par bloc, on utilise un registre qui contient la valeur de l'exposant à appliquer à un bloc de données. Cet exposant de bloc est constant pour un bloc de données. Chaque bloc de données est testé et mis à l'échelle par l'exposant de façon à éviter les débordements.

Le processeur travaillant sur des mots de N bits, la mantisse conserve N bits.

En fait, chaque mot est donc représenté par une mantisse sur N bits et un exposant, ou facteur d'échelle, qui est stocké dans un registre séparé, en général sur N bits. Les calculs se font en virgule fixe sur les mantisses du bloc, puis les résultats sont mis à l'échelle en fonction de l'exposant.

Cette représentation est utile quand N est petit (par exemple 16 bits) par rapport aux contraintes de dynamique et de précision du problème. Elle limite la perte de précision due à l'augmentation de la dynamique en virgule flottante, pour un nombre de bits fixé. La complexité des opérations reste raisonnable.

CHAPITRE 3

LES FAMILLES DE DSP TEXAS INSTRUMENTS

3.1 Les différentes familles et grandes classes de DSP

En 1982, Texas Instruments a lancé son premier DSP (*Digital Signal Processor*) : le TMS32010. Depuis, Texas Instruments s'est imposé comme le principal constructeur de processeurs de traitement de signal.

Le nom des DSP d'usage général de Texas Instruments commence par TMS320 : on oubliera donc souvent cet en-tête pour désigner un composant.

Les processeurs Texas Instruments sont organisés en familles qui correspondent à des classes de performances et d'applications.

3.1.1 Familles et classes de DSP standard

Les 6 familles les plus anciennes sont les suivantes :

- TMS320C1x, DSP format fixe;
- TMS320C2x, DSP format fixe;
- TMS320C5x, DSP format fixe;
- TMS320C3x, DSP format flottant;
- TMS320C4x, DSP format flottant;
- TMS320C8x, DSP multi-processeur.

Les DSP plus récents forment les familles :

- TMS320C54x, DSP format fixe;
- TMS320C20x, DSP format fixe;
- TMS320C24x, DSP format fixe;
- TMS320C62x, DSP format fixe à architecture VLIW;
- TMS320C67x, DSP format flottant à architecture VLIW.

Listing 8.13 – Programme IIR3.asm.

```
; Mise en cascade de 3 cellules
      .mmregs            ; utilisations de registres mappés en mémoire
in    .set  100h         ; adresse du port d'entrée
out   .set  101h         ; adresse du port de sortie
n     .set  3            ; nombre d'étages
; Réservation de la mémoire RAM
      .bss   wn,3*n      ; stockage des variables intermédiaires
      .bss   xn,1        ; signal d'entrée
      .bss   yn,1        ; signal de sortie
      .bss   tab,5*n     ; tableau des coefficients en mémoire data

coeff                    ; les coefficients sont en zone programme
      .word  5372        ; coefficient de mise à l'echelle de x(n), sc0
      .word  0           ; -b21 première section du premier ordre
      .word  5640        ; -b11
      .word  0           ; a21
      .word  12903       ; a11
      .word  12903       ; a01

      .word  -7585       ; -b22 deuxième section du second ordre
      .word  8581        ; -b12
      .word  2705        ; a22
      .word  1945        ; a12
      .word  2705        ; a02

      .word  -14144      ; -b23 troisième section du second ordre
      .word  6028        ; -b13
      .word  31851       ; a23
      .word  1377        ; a13
      .word  31851       ; a03

      .sect « vecteurs d'interruption »
RESET B debut            ; branchement au début du programme lors du RESET

Debut .text              ; section de programme
      SSBX  OVM          ; correction d'overflow
      SSBX  FRCT         ; décalage de 1 bit à gauche du produit
      SSBX  SXM          ; extension du signe
      STM   #tab, AR1    ; AR1 pointe la mémoire de data pour les coefficients
      RPT   #(5*n)
      MVPD  coeff,AR1+   ; transfert des coefficients de la mémoire
                         ; programme à la mémoire data
      STM   #wn,AR2      ; AR2 pointe les données intermédiaires
      RPTZ  #(3*n-1)
      STL   A,AR2+       ; mise à zéro des registres wn
      STM   #xn,AR5      ; AR5 pointe xn
      STM   #yn, AR3     ; AR3 pointe yn
```



```
      STM   #2, AR0            ; Offset dans le tableau des wn,
                               ; pour passer d'un étage au suivant.

Boucle :                       ; boucle sans fin de filtrage
      PORTR in,*AR5            ; lecture du signal d'entrée x(n)
      STM    #tab,AR4          ; AR4 pointe les coefficients
      STM    #wn+3*n-1, AR2    ; AR2 pointe le tableau des wn stockés du haut
                               ; en bas pour faciliter la mise a jour des wn
      MPY   *AR5, *AR4+, A     ; Mise a l'echelle de x(n) par sc0, stockage
                               ; dans A
      STM   #n-1               ; répétition de la boucle suivante n fois
      RPTB  Bcl-1
      ; calcul de la valeur intermédiaire
      MAC   *AR4+, *AR2-,A     ; A=x(n)sc0-w(n-2)*b2
      MAC   *AR4+, *AR2-,A     ; A=x(n)sc0-w(n-2)*b2-w(n-1)*b1
      STH   A, *AR2+0          ; ecriture de d(n) et AR2 pointe vers d(n-2)
                               ; calcul de la sortie de l'étage
      MPY   *AR4+,*AR2-,A      ; A=d(n-2)* a2
      MAC   *AR4+,*AR2,A       ; A=d(n-2)* a2+d(n-1)*a1
      DELAY *AR2-              ; d(n-1) copié dans d(n-2)
      MAC   *AR4+,*AR2,A       ; A=d(n-2)* a2+d(n-1)*a1+d(n)*a0
      DELAY *AR2-              ; d(n) copié dans d(n-1)
Bcl :                          ; fin de la boucle de calcul des 3 étages
      STH   A, *AR3            ; ecriture du résultat y(n) en mémoire
      PORTW *AR3, out          ; sortie vers le port du signal filtré
      B     Boucle             ; branchement au début pour le calcul
                               ; d'un nouvel échantillon filtré
```

8.5 Implantation de l'algorithme de Goertzel

8.5.1 Algorithme de détection de fréquence

La TFD (Transformée de Fourier Discrète) permet de détecter la présence d'un signal de fréquence particulière.

L'expression de la transformée de Fourier discrète pour un signal $x(n)$, échantillonné à la fréquence F_e, lorsque l'on considère N points de ce signal, est donnée par la formule suivante :

$$X(k) = \sum_{n=0}^{N-1} x(n) \cdot e^{-j\frac{2\pi k}{N}}$$

$X(k)$ donne la composante spectrale du signal $x(n)$ à la fréquence kF_e/N.

Cet algorithme nécessite le stockage de $N/2$ coefficients complexes et s'effectue en calculant N^2 opérations complexes pour toutes les raies spectrales.

Si l'on souhaite obtenir l'ensemble du spectre, il est possible d'effectuer le calcul par l'algorithme rapide de Cooley Tuckey (FFT) qui entraîne une réduction importante du nombre d'opérations à effectuer ($N\log_2(N)$ au lieu de N^2); le nombre de points N doit être une puissance de 2.

Lorsque la détection ne porte que sur quelques fréquences, l'algorithme de Goertzel permet d'effectuer le calcul de façon récursive et ne nécessite qu'un coefficient par fréquence sélectionnée, le nombre N de points peut être quelconque.

8.5.2 Principe de l'algorithme

On pose :

$$W_N = e^{-j\frac{2\pi}{N}} \text{ et } W_N^{kN} = W_N^{-kN} = 1$$

D'où

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{-k(N-n)}$$

Si l'on pose :

$$Y_k(m) = \sum_{n=0}^{m} x(n) W_N^{-k(m-n)}$$

on obtient alors :

$$X(k) = Y_k(N-1) W_N^{-k}$$

ainsi que la relation de récurrence :

$$Y_k(m) = x(m) + Y_k(m-1) W_N^{-k}$$

pour $m = 0$ à $N-1$

et $Y_k(0) = x(0)$

Cette relation est équivalente au filtre récursif :

$$H(z) = \frac{1}{1 - W_N^{-k} z^{-1}} = \frac{1 - W_N^k z^{-1}}{1 - 2\cos(\frac{2\pi k}{N}) z^{-1} + z^{-2}}$$

REMARQUE

Le module du pôle de ce filtre est égal à 1 et ce filtre est donc instable. Comme on le fait agir sur un nombre limité de points, on évitera cette instabilité.

8.5.3 Mise en œuvre de l'algorithme

On utilise une structure DN, c'est-à-dire faisant agir le dénominateur (noté $D(z)$) de la fonction de transfert puis le numérateur (noté $N(z)$), ce qui donne la relation de filtrage :

$$Y(z) = H(z)X(z) = D(z)X(z)$$

et $N(z) = 1 - W_N^k z^{-1}$

$$D(z) = \frac{1}{1 - 2\alpha_k z^{-1} + z^{-2}}$$

avec $\alpha_k = \cos(\frac{2\pi k}{N})$

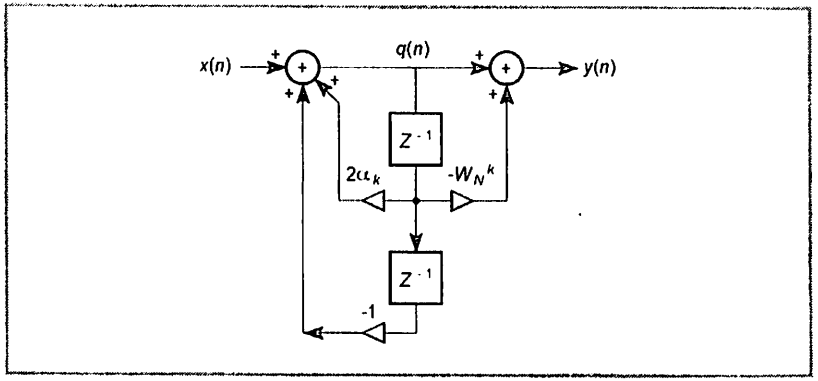

Figure 8.15 – Algorithme de Goertzel.

L'algorithme (*figure 8.15*) est le suivant :

$$q(0) = q(-1) = 0$$
$$q(n) = x(n) + 2\alpha_k q(n-1) - q(n-2)$$

pour $n = 1$ à N

et

$$y(n) = q(n) - W_N^k q(n-1)$$
$$X(k) = y(N)$$
$$X(k) = (q(N) - W_N^k q(N-1)) W_N^{-k}$$

Le calcul de $X(k)$ sera fait une seule fois lorsque la valeur de $q(N)$ et de $q(N-1)$ sera obtenue.

Calcul du module de la raie spectrale X(k)

$X(k)$ est un nombre complexe qui représente l'amplitude et la phase de la composante spectrale à la fréquence kF_e/N, la présence de cette fréquence dans le signal considéré sera déterminée par l'étude de la valeur du module de $X(k)$.

On a l'expression de ce module par :

$$|X(k)|^2 = |q(N) - W_N^k q(N-1)|^2$$

$$|X(k)|^2 = q(N)^2 + q(N-1)^2 - 2Re(W_N^k)q(N)q(N-1))$$

$$|X(k)|^2 = q(N)^2 + q(N-1)^2 - 2\alpha_k q(N)q(N-1))$$

8.5.4 Mise en œuvre du programme

La fréquence d'échantillonnage est choisie à 16 kHz, aussi si l'on veut détecter la présence de la fréquence 770 Hz, on choisira k = 36 et pour N = 748, on trouvera $\alpha_k = 0{,}95463$, ce qui correspond à une fréquence précise de 770,05Hz (kF_e/N).

Format de calcul et du coefficient α_k : les calculs doivent être exécutés en format fixe sur 16 bits et le coefficient α_k qui est compris entre – 1 et + 1 sera codé en format Q_{15}, ce qui donnera, pour 0,95463, un codage sur 16 bits de α_k égal à 31281.

On décide que le signal d'entrée $x(n)$ est compris entre – 1 et + 1 ; ainsi, dans le calcul temp = $xn + 2\alpha_k qn - qnn$, on codera l'ensemble en Q_{15} et le résultat $q(n)$ sera sauvé en Q_{15} pour plus de précision (voir *listing 8.14*).

On fait l'hypothèse que le gain du signal d'entrée est réglé pour éviter les saturations et on ne se préoccupera pas de les détecter.

Listing 8.14 – De la routine de détection de fréquence.

```
On suppose que le signal d'entrée est lu via un port d'entrée situé à
l'adresse 100h et que la raie spectrale est sortie vers un port situé à
l'adresse 101h.
; Programme Goertzel :
  .bss   qn, 2          ; reservation d'espace pour q(n-1) et q(n-2)
  .bss   xn,1           ; signal d'entrée
  .bss   W,1            ; raie spectrale
  .bss   alpha,1        ; coefficient alpha
alphap   .word 31281    ; coefficient cos(2Pi k/N) correspondant à 770Hz
                        ; pour Fe=16Khz
; en mémoire programme
N .set   748            ; nombre de points de calcul
                        ; AR3 pointe alpha
                        ; AR2 pointe q(n-2)
                        ; AR1 pointe x(n)
                        ; Initialisation des modes du DSP
SSBX     FRCT           ; décalage du produit pour un cadrage en Q15
SSBX     SXM            ; extension du signe lors de décalages
RSBX     OVA            ; mise à 0 des flag d'overflow
                        ; des accumulateurs A et B

RSBX     OVB ;
                        ; Initialisation des AR
LD       #qn, AR2
LD       #xn, AR1
LD       #W, AR4
LD       #alpha, AR3
MVPD     alphap, *AR3   ; transfert de alpha de la mémoire programme
                        ; à la mémoire data
                        ; initialisation de q(0) et q(-1) à zéro
RPTZ     #1
STL      A, *AR2+       ; Q(0)=0 et q(-1)=0
MAR      *AR2-          ; AR2 pointe q(n-2)
STM      #N-1, BRC      ; répétition de l'algorithme N fois
RPTB     Bcle-1
PORTR    100h, *AR1     ; lecture de x(n) et écriture en mémoire
LD       *AR1,16, A     ; AH=x(n)
SUB      *AR2-, 16, A, B; A= x(n)- q(n-2) et AR2 pointe q(n-1)
MAC      *AR2, *AR3, A  ; A=x(n)-q(n-2)+alpha q(n-1)
MAC      *AR2, *AR3, A  ; A=x(n)-q(n-2)+2.alpha q(n-1)
DELAY    *AR2           ; q(n-1) copié dans q(n-2) pour l'itération
                        ; suivante
STH      A, *AR2+       ; écriture de q(n) à la place de q(n-1).
Bcle                    ; Fin des N itérations
                        ; Calcul de l'energie de la raie
                        ; W=q(N)*q(N)-2 alpha q(N)*q(N-1)+ q(N-1) q(N-1)
LD       *AR2, 16, A    ; A=q(N-1)
MPYA     *AR2-          ; B=q(N-1)*q(N-1)  T=q(N-1)
MPY      *AR2, A        ; A=q(N)* q(N-1)
LD       *AR3, T        ; T=alpha
MPYA     A              ; A=alpha *q(n)* q(N-1)
SUB      A, 1, B        ; soustraction avec décalage de 1 bit
                        ; équivalente à 2 fois alpha*q(N)*q(N-1)
                        ; B=q(N-1)²-2 alpha *q(n)* q(N-1)
LD       *AR2, T        ; T=q(N)
MAC      *AR2, B        ; B=q(N-1)²-2 alpha *q(n)* q(N-1)+q(N) ²
STH      B,*AR4         ; Sauvegarde de l'energie
PORTW *AR4, 101h        ; Sortie vers le port de l'energie
FIN
NOP
B FIN                   ; fin de la routine, attente d'interruption
```

8.6 Implantation d'un algorithme de Viterbi, principes et exemples d'applications avec le simulateur *Code Composer*

L'algorithme de Viterbi est un algorithme de programmation dynamique qui détermine la séquence de transitions d'états la plus probable dans un diagramme d'états, étant donnée une séquence bruitée de symboles ou l'observation d'un signal.

Les applications sont nombreuses. On peut citer par exemple le décodage des codes correcteurs convolutionnels ou l'égalisation des canaux de transmission, la reconnaissance de la parole par des méthodes de type HMM (*Hidden Markov Models*).

On va s'intéresser uniquement au décodage des codes convolutifs, mais la présentation peut se généraliser simplement aux autres applications.

8.6.1 Codage convolutionnel

Il existe 2 grandes familles de codes correcteurs d'erreur, les codes en blocs et les codes convolutionnels.

Pour les codes en bloc, à chaque paquet de k bits (ou symboles) d'information, on associe un mot de code formé de n bits (ou symboles), $n > k$, contenant une certaine redondance, qui pourra être exploitée en réception pour détecter ou corriger les erreurs. On se limite ici au cas binaire. On transmet ce mot de code au lieu des k bits d'information. Le débit binaire est donc multiplié par le rapport n/k. On appelle n la longueur, k la dimension et $R = K/N$ le taux du code. Un paramètre important du code est sa distance minimale, c'est-à-dire la plus petite distance de Hamming entre 2 mots de code différents. On appelle distance de Hamming de 2 mots binaires, le nombre de bits différents entre ces 2 mots. Par exemple la distance de Hamming entre les 2 mots 110 et 101 vaut 2.

Pour les codes convolutionnels, la séquence binaire codée correspondant à la séquence d'informations originale ne s'obtient plus en codant des blocs successifs de k bits indépendamment les uns des autres, mais en appliquant une sorte de convolution logique à la séquence d'entrée, d'où le nom de codes convolutionnels. La convolution s'obtient par un registre à décalage dont certaines sorties sont ajoutées modulo 2 (ce qui est équivalent à une porte « OU EXCLUSIF » souvent appelée XOR).

Plus précisément, pour un code convolutionnel de taux $1/n$, on utilise un registre à décalage de longueur K et on génère n sorties par addition modulo 2 de certaines sorties du registre à décalage. Ainsi, à chaque bit d'entrée correspond n bits de sortie. Les n bits sont multiplexés en sortie et le débit est multiplié par n. La *figure 8.16* représente un codeur convolutionnel de taux $1/n = 1/3$.

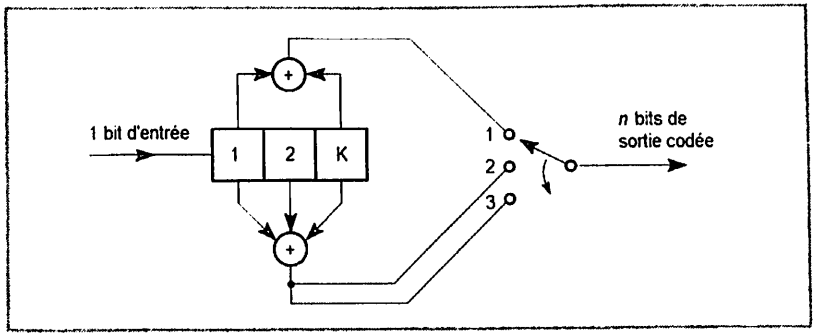

Figure 8.16 – Codeur convolutionnel de taux 1/n = 1/3 et de longueur de contrainte K = 3.

La longueur K du registre à décalage est appelée la longueur de contrainte. Chaque bit de sortie dépend au plus de K bits d'entrée successifs.

On peut écrire les équations liant les n bits de sortie $y_i(t)$ ($i \in [1, n]$) au bit d'entrée $x(t)$ sous la forme d'une convolution avec des additions modulo 2, entre l'entrée $x(t)$ et une réponse impulsionnelle $h_i(t)$ de longueur finie et de coefficients $h_i(l)$ égaux à 1 ou 0 selon que la i^e case du registre à décalage est connectée ou non au XOR. Pour l'exemple de la *figure 8.10*, on obtient :

$$y_1(t) = x(t) \oplus x(t-2)$$
$$y_2(t) = x(t) \oplus x(t-1) \oplus x(t-2)$$
$$y_2(t) = x(t) \oplus x(t-1) \oplus x(t-2)$$

Pour un code convolutionnel de taux k/n, à chaque paquet de k nouveaux bits en entrée, correspondent n nouveaux bits en sortie. On utilise K registres à décalage de longueur k connectés entre eux et n bits sont générés en sortie pour chaque paquet de k bits en entrée. Les bits de sortie sont obtenus par addition modulo 2 de certaines sorties des registres. Les bits d'entrée sont décalés de k dans les registres. La *figure 8.17* représente un codeur convolutionnel de taux $k/n = 2/3$. Le nombre K de registres à décalage de k bits est la longueur de contrainte du code.

On peut caractériser les codes convolutionnels par un ensemble de n vecteurs de Kk bits. À chaque bit de sortie correspond un vecteur. Ce vecteur représente les sorties du registre à décalage qui sont additionnées pour fabriquer le bit de sortie. Le j^e bit vaut 1 si la j^e sortie du registre à décalage est connectée à l'additionneur. Ainsi, le codeur de la *figure 8.16* est caractérisé par les 3 vecteurs : 101 pour le bit de sortie n° 1, 111 pour le bit n° 2 et 111 pour le bit n° 3. On note généralement en octal ces vecteurs, ce qui donne ici 5, 7, 7. Pour l'exemple de la *figure 8.17*, les 3 vecteurs sont 17, 6, 15 en octal. On parle parfois de fonctions génératrices pour désigner ces n vecteurs.

Une caractéristique importante des codes convolutifs est leur distance libre dont dépend leur capacité de correction d'erreurs. La distance libre est la plus petite distance non nulle entre 2 séquences codées.

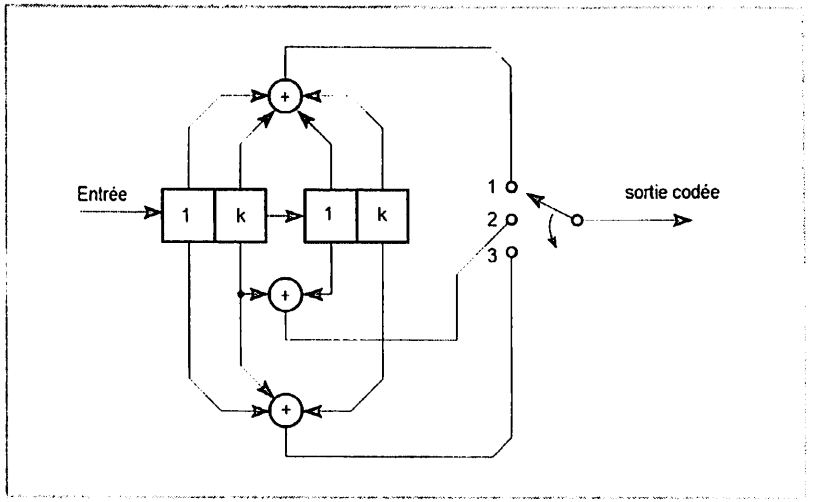

Figure 8.17 – Codeur convolutionnel de taux k/n = 2/3 et de longueur de contrainte K = 2.

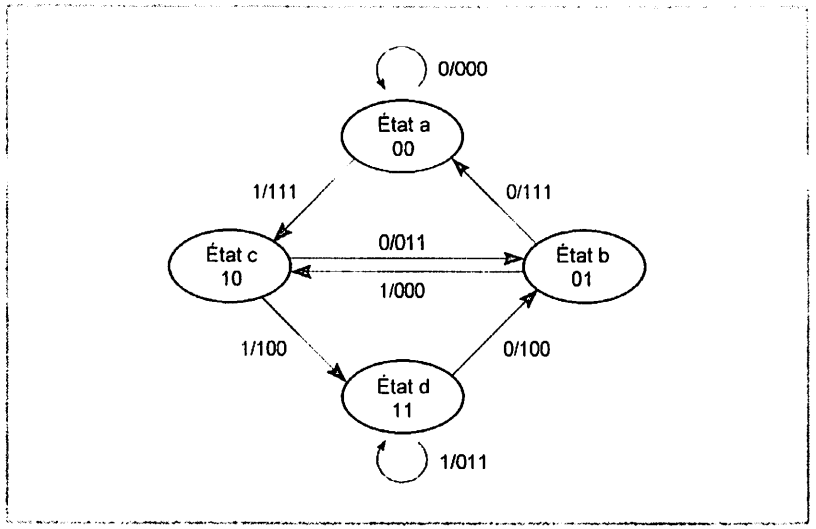

Figure 8.18 – Diagramme d'état du codeur de la figure 8.16.

La sortie d'un codeur convolutif dépend de son entrée et du contenu de $K-1$ registres de longueur k. On peut considérer que ces codeurs sont des machines d'état finies avec $2^{k(K-1)}$ états différents. On peut de ce fait les représenter de différentes manières, par des graphes d'états, des arbres, ou des treillis. La *figure 8.18* représente le diagramme d'état du code de la *figure 8.16*. La longueur de contrainte de ce code vaut 3, il a donc $4 = 2^2$ états. De chaque état peuvent partir $k = 2^1$ transitions, selon que l'entrée vaut 1 ou 0, et à chaque état peuvent arriver de même $2 = 2^k$ flèches. Chaque état correspond aux $k(K-1)$ bits du registre à décalage qui seront conservés pour une nouvelle entrée de k bits. Dans l'exemple, les 4 états correspondent aux dibits 00, 01, 10 et 11.

Dans la *figure 8.18*, on a indiqué sur chaque flèche de transition d'état la valeur du bit en entrée et des 3 bits de sortie séparés par une barre de division.

Dans les diagrammes en arbres ou en treillis, on représente toutes les évolutions possibles au cours du temps des états, des entrées et des sorties du codeur. On va présenter le diagramme treillis qui est celui utilisé par l'algorithme de Viterbi. La *figure 8.19* représente le diagramme treillis associé au codeur de la *figure 8.16* et cela pour une durée d'émission de 5 bits et en partant de l'état initial 00.

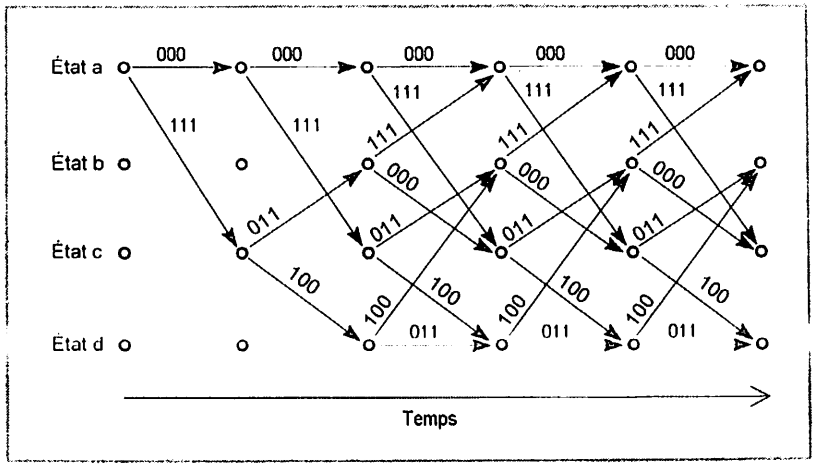

Figure 8.19 – Diagramme treillis du codeur de la figure 8.16.

Dans ce diagramme, on représente à chaque instant (nouveau bit d'information en entrée) les 4 états possibles avec les transitions arrivant et partant de chaque état. Chaque nœud correspond à un état. On a représenté l'émission d'un zéro par une flèche descendante et celle d'un 1 par une flèche montante. Après le deuxième étage du treillis, le treillis est stationnaire, tous les étages sont ensuite identiques avec 2 branches entrant et 2 branches sortant à chaque nœud. Chaque étage correspond à un intervalle de temps pour l'émission d'un bit d'information

ou de n bits codés (on appelle parfois symbole l'ensemble de ces n bits codés). On a représenté au-dessus de chaque branche la séquence de 3 bits émise lors de cette transition.

Pour un code de taux k/n avec une longueur de contrainte K, le nombre d'états du codeur est $2^{k(K-1)}$. Le treillis a donc $2^{k(K-1)}$ nœuds par étage avec 2^k branches entrant et sortant de chaque nœud.

8.6.2 Principe de l'algorithme de Viterbi pour le décodage des codes convolutionnels

Deux situations se présentent en pratique, soit les données sont transmises en trames de longueur finie, chaque trame étant codée indépendamment des autres, soit les données sont codées de façon continue aussi longtemps que dure la communication. La mémoire d'un codeur convolutionnel est infinie.

Dans la première situation, qui est celle de la transmission TDMA du GSM par exemple, on ajoute généralement à chaque trame $K-1$ bits de queue (*tail bits*) égaux à 0, avant le codage. On impose souvent par ailleurs que l'état de départ soit nul. Le décodeur connaît alors les états d'arrivée et de départ (nuls), ce qui est utile au décodage. Dans cette situation, le décodeur peut attendre d'avoir reçu la séquence codée complète avant de la décoder, bien que parfois il se comporte comme on va le voir pour la deuxième situation.

Dans la deuxième situation, le décodeur ne peut pas attendre la fin de la séquence, le retard introduit serait trop grand et l'algorithme nécessiterait trop de mémoire. Aussi, le décodeur prend-il une décision avec un retard maximum D. Si ce retard est supérieur à environ 5 fois la longueur de contrainte, la dégradation, introduite par la limitation du retard, est négligeable en terme de probabilité d'erreur.

Par la suite, on note N la durée totale de la séquence reçue sur laquelle il faut prendre une décision.

Le but de l'algorithme de Viterbi est de trouver la séquence émise la plus vraisemblable parmi toutes les séquences possibles de longueur N, étant donnée la séquence reçue. Une fois la séquence émise connue, on en déduit la séquence d'états correspondante et la séquence binaire originale non codée. On verra que cette notion de vraisemblance peut se ramener à une notion de distance, que l'on précisera plus loin, entre la séquence reçue et les séquences possibles. Par la suite, on considérera donc que le décodeur cherche parmi les séquences possibles la séquence la plus proche de la séquence reçue au sens d'une certaine distance.

Pour un treillis à $E = 2^{k(K-1)}$ états, avec 2^k branches sortant de chaque nœud, et une durée de séquence N, il y a $E \cdot 2^{kN}$ chemins possibles, c'est-à-dire séquences possibles à tester, nombre qui augmente de manière géométrique avec N. Par exemple pour $R = 1/2$, $K = 7$ et $N = 35$, le nombre de chemins possibles est égal à 64×2^{35}, ce qui est supérieur à 10^{12}. Heureusement, l'algorithme de Viterbi va

permettre de résoudre le problème avec une complexité proportionnelle à N, ainsi qu'au nombre d'états E et à 2^k.

En effet, pour trouver la meilleure séquence, il n'est pas nécessaire de toutes les tester exhaustivement. Considérons un nœud i à l'étape n dans le treillis : 2^k chemins arrivent en ce nœud. On peut calculer les 2^k distances entre la séquence reçue de l'instant 1 à l'instant n et les séquences de ces 2^k chemins de longueur n. On peut, pour la suite des opérations, éliminer $(2^k - 1)$ chemins et ne conserver que celui qui donne la plus petite distance. En effet, si la fin de la meilleure séquence globale part de ce nœud là, le début de cette meilleure séquence passe forcément par le chemin qui donne la plus petite distance pour arriver à ce nœud. On appelle ce chemin le « survivant ».

On peut appliquer ce principe de manière itérative à chaque étage du treillis. Ainsi, à l'étape t dispose-t-on pour les E nœuds d'entrée des E meilleurs chemins arrivant en ces nœuds ainsi que des distances associées. On note $D_i(t)$ ces distances, où i représente le numéro du nœud et t le numéro de l'étape. On peut calculer, pour chaque nœud de sortie, les distances des 2^k chemins arrivant en ces nœuds. La distance du chemin arrivant au nœud j à l'instant $t+1$ en provenance du nœud i à l'instant t, notée $D_{i,j}(t+1)$ peut s'écrire :

$$D_{i,j}(t+1) = Di(t) + d_{i,j}(t)$$

où $d_{i,j}(t)$ est la distance sur la branche reliant le nœud i au nœud j à l'étape t. On appelle souvent $D_{i,j}(t+1)$ la distance cumulée du chemin, et $d_{i,j}(t)$ la distance locale de la branche i, j.

Le chemin survivant au nœud j est celui qui donne la distance minimale. Le numéro l du nœud d'origine du chemin survivant, arrivant au nœud j à l'instant $t+1$ est donné par :

$$l = \arg\ \min_i\ (D_{i,j}(t+1))$$

La distance correspondant à ce chemin est $D_j(t+1)$.

À chaque étape, on calcule et on garde en mémoire les E distances $D_j(t+1)$ ainsi que les nœuds d'origine l des survivants. On fait ces calculs sur les N étages. On fait donc $N \times E \times 2^k$ calculs de distance et on doit garder en mémoire $2E$ distances cumulées, 2^kE distances locales et NE indications de numéro de nœuds.

À la fin, après N étapes, soit on sait quel est l'état d'arrivée final (parce que l'émetteur a ajouté des *trailing* bits), soit on cherche quel état final correspond à la plus petite distance. Puis, à partir de cet état final retenu, on repart en sens inverse pour tracer la meilleure séquence globale qui aboutit à ce nœud final. On peut le faire car à chaque étape et à chaque nœud, on a mémorisé l'origine du meilleur chemin arrivant en ce nœud. On appelle *traceback* ce dernier travail.

La *figure 8.20* illustre l'étape de calcul de distances pour une étape du treillis du codeur de la *figure 8.16*. On a utilisé une distance de Hamming entre la séquence reçue et les séquences possibles.

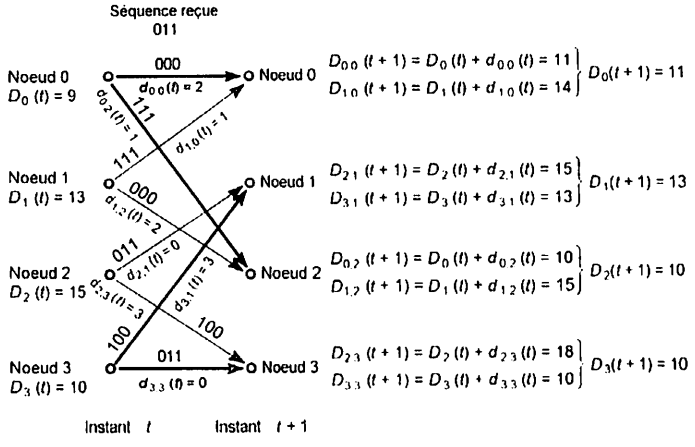

Figure 8.20 – Algorithme de Viterbi, calcul des distances sur une étape du treillis, calcul du survivant.

La *figure 8.21* illustre la procédure de *traceback* pour le codeur de la *figure 8.16*. On a choisi $N = 4$, pour obtenir un dessin pas trop complexe et on a utilisé une distance de Hamming entre la séquence reçue et les séquences possibles. On a indiqué les distances locales de chaque branche en dessous de la branche.

Sur la *figure 8.21* les traits épais correspondent aux chemins survivants (parfois il y en a 2 en un nœud parce que les distances cumulées sont égales).

La distance globale minimale à l'instant 4 est celle obtenue au nœud 3. Elle vaut $D_3(4) = 1$. La procédure de *traceback* part donc du nœud numéro 3 à l'instant 4 et remonte vers l'origine en suivant les chemins survivants. Le meilleur chemin global est représenté sur la figure avec des petites flèches qui retournent vers l'origine du treillis. La suite des nœuds rencontrés sur le chemin optimal aux instants $t = 4, 3, 2, 1$ est la suite n° 3, 3, 2, 0, 0. On peut ensuite remettre cette séquence de nœuds dans le bon ordre. La séquence de nœuds du chemin optimal rangés par ordre de temps croissant est donc : 0, 0, 2, 3, 3. La séquence codée émise le long de ce chemin est : 000 111 100 011, ce qui correspond à la suite de bits d'information non codée : 0 1 1 1.

En ce qui concerne les distances utilisées dans la méthode, on distingue 2 cas en pratique, correspondants aux situations de décodage appelées *hard* ou *soft*.

On parle de décodage *hard* quand le récepteur convertit les signaux reçus à chaque période bit en 1 et 0. Dans ce cas-là, on utilise la distance de Hamming pour mesurer la distance entre 2 séquences de bits. On peut montrer que si la proba-

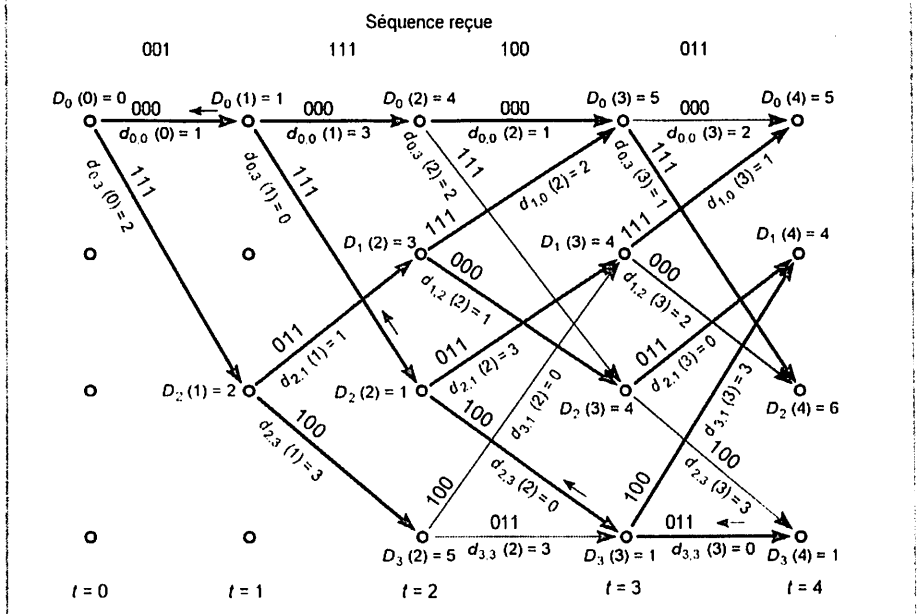

Figure 8.21 – Algorithme de Viterbi, procédure de traceback.

bilité d'erreur bit du canal est inférieure à 0,5, la distance de Hamming correspond bien à rechercher la séquence possible la plus probable compte tenu de la séquence reçue, c'est-à-dire la plus vraisemblable.

On parle de décodage *soft* quand le récepteur ne force pas à 1 ou 0 le signal reçu, mais donne à chaque période bit la valeur de sortie r du filtre adapté du récepteur. Typiquement, cette valeur est proche de 1 si on a émis un 1 et proche de – 1 si on a émis un zéro. Elle dépend non seulement du bit émis mais aussi du bruit du canal. On va écrire les relations pour un code de taux $1/n$. À chaque étape, on reçoit des paquets de n échantillons notés $r_0(t)\ldots, r_i(t), r_{n-1}(t)$. Les échantillons de la séquence r sont appelés soft bits. On note $c_0(t)\ldots, c(t), c_{n-1}(t)$ une séquence possible émise par le codeur à l'étape t. Si le bruit additif est blanc et gaussien, la vraisemblance d'une séquence s'écrit simplement car les échantillons reçus r sont gaussiens et indépendants entre eux. On considère la séquence totale reçue entre l'instant 0 et l'instant N et on la note R. On note par ailleurs $C^{(i)}$ une des séquen-

– lire et tester ce bit utile correspondant ;

– en déduire le numéro du nœud précédent sur le meilleur chemin.

Le *listing 8.16* donne le code pour une procédure de *traceback*.

Listing 8.16 – Code C54x pour la procédure de traceback.

```
* On utilise AR2 pour pointer sur la table des mots de transition.
* On appelle adr_fin_TRN l'adresse de fin de ce buffer
* On utilise AR2 pour pointer sur le buffer contenant
* les bits d'information décodés avec un bit par mot.
* On appelle adr_fin_bit l'adresse de fin de ce buffer
* N est le nombre de bits à décoder, nombre d'étapes de treillis
* K est la longueur de contrainte
* On suppose que le nœud final est le nœud 0 (tail bits)
* Dans A se trouve la valeur d'un numéro de nœud sur le chemin optimal

                STM   #adr_fin_TRN, AR2
                STM   #adr_fin_bit, AR1
                STM   #(N-1), AR4
* AR0 contient le numéro du mot de transition dans
* un bloc de 2^(K-5)mots. Il est utilisé en offset par rapport à AR2
NB_WTRN         .set  2^(K-5)
OFFSET          .set  2^(K-2)+1
NBEsur2         .set  2^(K-2)
MASQUE          .set  2^(K-2)-1

                LD    #0, A
* 2 boucles imbriquées,
* l'externe liée à AR4 compte jusqu'à N, l'interne RPTB jusqu'à 16
                STM   #15,BRC
boucleN         RPTB  fin-1
* Calcul du bit décodé par test du bit K-2 de A
                STM   #0,*AR1
                SUB   #NBEsur2, A,B
                XC    2, BGEQ
                STM   #0,*AR1-
* Calcul du numéro du mot de transition
                AND   #MASQUE, A,B
                SFTL  B,-3
                STLM  B, *(AR0)
                MAR   *+(-2^(K-5))
                MAR   *AR2+0    ; AR2 pointe sur le bon mot
                                ; de transition
* Calcul du numéro de bit utile
                SFTL  A, -(K-2), B
                AND   #1,B
                ADD   A, 1, B
                STLM  B, T
                BITT  AR2-0     ; on met le bit utile dans T
                                ; et on réinitialise AR2
                ROLTC A         ; on a dans A le nouvel état
                                ; = le numéro du nœud précédent
                                ; sur le meilleur chemin
fin
                BANZD boucleN,*AR4-
                STM   #15,BRC
```


8.7 Exemple complet d'application : modulateur GMSK, implantation sur simulateur (*Code Composer*)

Dans cet exemple, nous allons étudier un modulateur GMSK. La modulation GMSK est la modulation utilisée en particulier sur les liaisons GSM de communications mobiles ainsi que dans les systèmes DECT de téléphonie sans cordon. On verra comment implanter le modulateur, comment générer une fréquence pure, comment utiliser les possibilités de tracés de diagramme de l'œil du simulateur *Code Composer*.

8.7.1 Principe de la modulation GMSK

Le sigle GMSK signifie *Gaussian Minimum Shift Keying*. Il s'agit d'une modulation de fréquence binaire à phase continue d'indice 1/2, dont l'occupation spectrale est limitée par l'utilisation d'un filtre gaussien sur les données. Selon que l'on émet un 0 ou un 1, la fréquence instantanée du signal modulée varie autour d'une fréquence centrale f_c.

Soit une suite binaire a_k valant + 1 ou – 1, et $x(t)$ le signal modulé en fréquence à phase continue d'indice h. Par définition le signal $x(t)$ s'écrit :

$$x(t) = \cos(2\pi f_c t + \Phi(t)) \quad \text{avec}$$
$$\text{pour } t \in [nT,(n+1)T]$$
$$\Phi(t) = 2\pi h \int_{-\infty}^{t} \sum_{k=-\infty}^{n} a_k s(\tau - kT)\mathrm{d}\tau$$

T est la durée d'un bit. La fonction $s(t)$ est normalisée par :

$$\int_{-\infty}^{+\infty} s(\tau)\mathrm{d}\tau = \frac{1}{2}$$

On note généralement $q(t)$ la primitive de la fonction $s(t)$:

$$q(t) = \int_{-\infty}^{t} s(\tau)\mathrm{d}\tau$$

On appelle impulsion élémentaire de phase la fonction $2\pi hq(t)$ notée $\varphi(t)$, et on écrit le signal $x(t)$ en fonction de $q(t)$ sous la forme :

pour $t \in [nT,(n+1)T]$

$$\Phi(t) = 2\pi h \sum_{k=-\infty}^{n} a_k q(t-kT) = \sum_{k=-\infty}^{n} a_k \varphi(t-kT)$$

$$x(t) = \cos\left(2\pi f_c t + \Phi(t)\right) = \cos\left(2\pi f_c t + 2\pi h \sum_{k=-\infty}^{n} a_k q(t-kT)\right)$$

$$= \cos\left(2\pi f_c t + \sum_{k=-\infty}^{n} a_k \varphi(t-kT)\right)$$

avec :

$$\varphi(t) = 2\pi h \int_{-\infty}^{t} s(\tau)\mathrm{d}\tau$$

$$\varphi(t) = 0 \quad \forall t < 0$$

$$\varphi(t) = h\pi \quad \forall t > LT$$

Dans une modulation GMSK, h vaut 1/2 et la fonction $s(t)$ est égale à un rectangle de durée T convolué avec une gaussienne $h(t)$ appelée filtre gaussien. La transformée de Fourier $H(f)$ de la fonction gaussienne est une gaussienne en fréquence. $H(f)$ est un filtre passe-bas. On note B la largeur de bande à 3 dB du filtre gaussien.

Les modulations GMSK diffèrent par la valeur de leur produit BT, où B est la largeur de bande du filtre gaussien et T la durée d'un bit. Pour la norme GSM, BT vaut 0,3, pour la norme DECT, BT vaut 0,5. Plus BT est petit, plus le signal modulé a un spectre étroit mais plus la démodulation est difficile.

L'expression de $h(t)$ et de $H(f)$ est la suivante :

$$h(t) = \sqrt{\frac{2\pi}{\ln(2)}} B \exp\left(-\frac{2\pi^2 B^2}{\ln(2)} t^2\right)$$

$$H(f) = \exp\left(-\frac{\ln(2)}{2B^2} f^2\right)$$

En théorie, la durée de la gaussienne est infinie, mais en pratique, on limite la fonction $h(t)$ aux quelques périodes bit sur lesquelles elle est significativement non nulle. Cette durée est inversement proportionnelle à B. Pour un produit BT de 0,3, on considère en général que $h(t)$ est non nulle sur une période de 3 ou 4 bits. Et pour un BT de 0,5, on considère que $h(t)$ est non nul sur une durée de 2 bits seulement. La convolution de $h(t)$ avec un rectangle de durée T dure un bit de plus, on notera LT bits la durée de la gaussienne plus une période bit.

L'impulsion élémentaire de phase $\varphi(t)$ varie pendant LT bits, puis reste constante et égale à $\pi/2$.

La *figure 8.25* représente l'impulsion élémentaire de phase pour différentes valeurs du produit BT sur une durée de 5 bits.

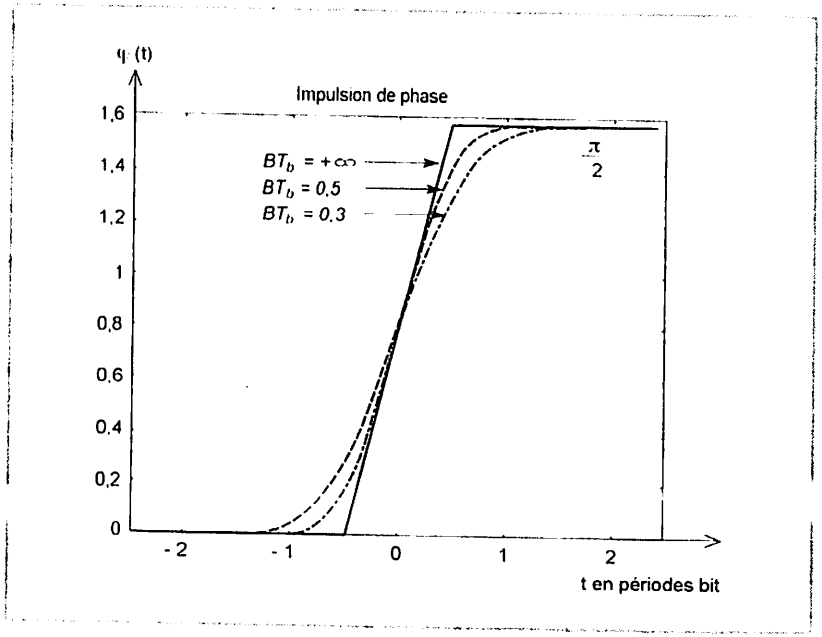

Figure 8.25 – Impulsion élémentaire de phase pour différentes valeurs du produit BT.

Pour réaliser une modulation GMSK avec un DSP, on utilise souvent une modulation dite en quadrature. Le signal modulé $x(t)$ peut s'écrire :

$$x(t) = \cos\left(2\pi f_c t + \Phi(t)\right) = \cos\left(\Phi(t)\right)\cos\left(2\pi f_c t\right) - \sin\left(\Phi(t)\right)\sin\left(2\pi f_c t\right)$$

$$x(t) = x_I(t)\cos\left(2\pi f_c t\right) - x_Q(t)\sin\left(2\pi f_c t\right)$$

Les signaux $x_I(t)$ et $x_Q(t)$ modulent en amplitude les 2 porteuses en quadrature cos $(2\pi f_c t)$ et sin $(2\pi f_c t)$. Ces 2 signaux sont en général générés par un DSP, alors que les porteuses de fréquence élevée sont générées de manière analogique. Dans l'exemple, on verra comment générer une fréquence porteuse basse, la génération de fréquences pures étant utile dans de nombreuses applications.

La *figure 8.26* représente une suite binaire et le signal modulé GMSK correspondant pour une porteuse $f_c = 2/T$. Sur cette figure, on a de plus représenté la phase $\Phi(t)$ en radians ainsi que les signaux $x_I(t) = \cos(\Phi(t))$ et $x_Q(t) = \sin(\Phi(t))$. Les signaux $x_I(t)$ et $x_Q(t)$ sont respectivement appelés composante ou signal en phase et en quadrature. On parle aussi parfois de signaux bande de base.

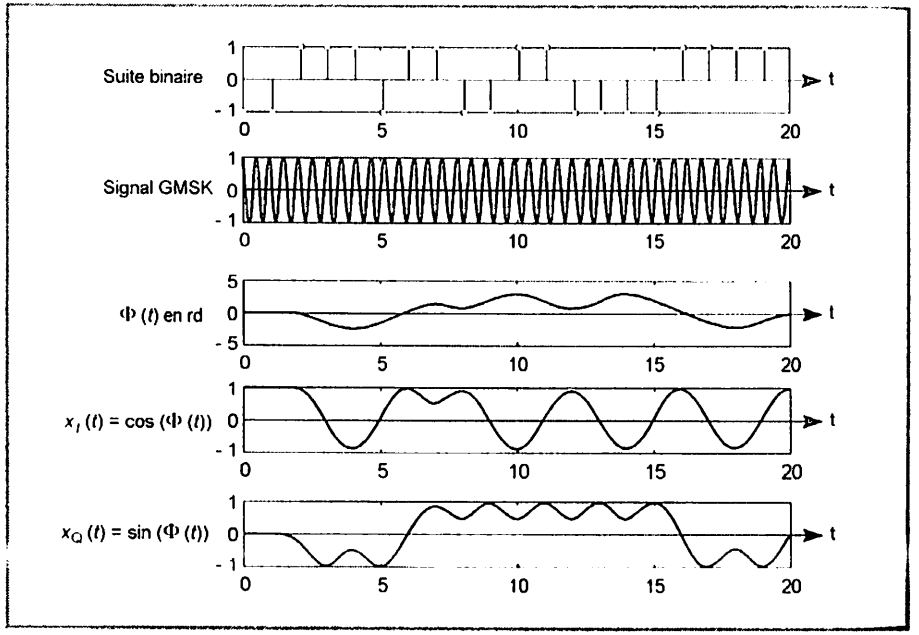

Figure 8.26 – Signal modulé GMSK, signal de phase et signaux x_I, x_Q associés.

8.7.2 Implantation d'un modulateur GMSK sur DSP

Pour implanter un modulateur GMSK sur DSP, il faut générer la phase $\Phi(t)$ et en déduire par lecture en table les signaux $x_I(t) = \cos(\Phi(t))$ et $x_Q(t) = \sin(\Phi(t))$. Pour l'exemple on générera aussi les 2 porteuses en quadrature cos $(2\pi f_c t)$ et sin $(2\pi f_c t)$, avec $fc = 1/T$. On choisit ici d'utiliser un rapport T/T_e égal à 8, c'est-à-dire que l'on a 8 échantillons par bit, et une valeur de fréquence $f_c = 1/T$.

Calcul des phases

On calcule la partie évolutive de la fonction élémentaire de phase $\varphi(t)$ de durée LT en utilisant la formule vue précédemment. On peut utiliser un logiciel, comme Matlab par exemple, pour faire ce calcul. Pour BT = 0,3, on prend L = 4 et pour BT = 0,5 L = 3.

Les lignes suivantes donnent la suite des 32 valeurs de la partie évolutive de $\varphi(t)$ pour BT = 0,3, L = 4, et un rapport $T/T_e = 8$. On appelle *phi*03 cette variable de 32 échantillons.

*phi*03 = [0,0001 0,0002 0,0005 0,0012 0,0028 0,0062 0,0127
0,0246 0,0446 0,0763 0,1231 0,1884 0,2740 0,3798
0,5036 0,6409 0,7854 0,9299 1,0672 1,1910 1,2968
1,3824 1,4476 1,4945 1,5262 1,5462 1,5581 1,5646
1,5680 1,5696 1,5703 1,5706]

Les lignes suivantes donnent la suite des 24 valeurs de la partie évolutive de $\varphi(t)$ pour BT = 0,5, L = 3, et un rapport $T/T_e = 8$. On appelle *phi*05 cette variable de 24 échantillons.

*phi*05 = [0,0000 0,0001 0,0003 0,0013 0,0048 0,0148 0,0386
0,0860 0,1661 0,2823 0,4310 0,6025 0,7854 0,9683
1,1398 1,2885 1,4047 1,4848 1,5322 1,5560 1,5660
1,5695 1,5705 1,5707]

La dernière valeur des tableaux *phi*03 et *phi*05 est presque égale à $\pi/2$. Après sa partie évolutive de L bits, la fonction élémentaire de phase est constante et égale à $\pi/2$.

Pour calculer la phase $\Phi(t)$, on traite à part la partie évolutive de $\varphi(t)$ de durée LT et sa partie constante de durée infinie. La phase $\Phi(t)$ peut s'écrire, pour $t \in [nT, (n+1)T]$:

$$\Phi(t) = \sum_{k=-\infty}^{n} a_k \varphi(t - kT) = \sum_{k=-\infty}^{n-L} a_k \varphi(t - kT) + \sum_{k=n-L+1}^{n} a_k \varphi(t - kT)$$

avec :

$$\varphi(t) = 0 \quad \forall t < 0$$

$$\varphi(t) = h\pi = \frac{\pi}{2} \quad \forall t > LT$$

Et comme $\varphi(t)$ est constante égale à $\pi/2$ pour $t > LT$, le premier terme de la somme est constant dans l'intervalle $[nT, (n+1)T]$. On va le noter *phimem* (mémoire de phase). Finalement, les équations que l'on va implanter sont les suivantes, pour $t \in [nT, (n+1)T]$:

$$\Phi(t) = \sum_{k=-\infty}^{n-L} a_k \varphi(t - kT) + \sum_{k=n-L+1}^{n} a_k \varphi(t - kT) = \frac{\pi}{2} \sum_{k=-\infty}^{n-L} a_k + \sum_{k=n-L+1}^{n} a_k \varphi(t - kT)$$

$$\Phi(t) = phimem(n) + \sum_{k=n-L+1}^{n} a_k \varphi(t - kT)$$

$$phimem(n) = phimem(n-1) + a(n-L)\frac{\pi}{2}$$

On implantera ces 2 équations de façon récursive. On appelle phase la variable contenant les valeurs $\Phi(t)$. On appelle *phi* le tableau contenant la partie évolutive de $\varphi(t)$, tableau qui contient $Nphi = LTT_e$ échantillons. On note *NS* le nombre d'échantillons par bit, c'est-à-dire TT_e. On appelle an la suite binaire égale à + 1 ou – 1. On va calculer la phase $\Phi(t)$ (variable phase) puis obtenir par lecture en table le cosinus et le sinus de cette phase pour fabriquer les signaux $x_I(t)$ et $x_Q(t)$. On traite à part les *L* premiers intervalles bit pour lesquels phimem est nul. Puis, à partir du bit *L* + 1, on tient compte de *phimem* qu'on initialise à $an(1)\pi/2$.

On effectue le traitement suivant :

De $i = 1$ jusqu'à $i = L$

  de $j = (i-1)Ns + 1$ jusqu'à $j = (i-1)Ns + Nphi$

    phase$((i-1)Ns + j)$ = phase$((i-1)Ns + j)$ + $phi(j)$an(i)

  finde

finDE

$phimem = pil/2$ an(1)

De $i = L + 1$ jusqu'au dernier bit

  de $j = (i-1)Ns + 1$ jusqu'à $j = (i-1)Ns + Nphi$

    phase$((i-1)Ns + j)$ = phase$((i-1)Ns + j)$ + $phi(j)$an(i)

  finde

  de $j = (i-1)Ns + 1$ jusqu'à $j = i\,Ns$

    phase$((i-1)Ns + j)$ = phase$((i-1)Ns + j)$ + $phimem$

    xi = cos (phase$(i-1)Ns + j)$

    xq = sin (phase$((i-1)Ns + j)$

  finde

  $phimem = phimem + \frac{\pi}{2}$ an$(i + 1 - L)$

finDE

Calcul des cosinus et des sinus par lecture en table

Pour générer les valeurs de sinus et de cosinus, on utilise dans cet exemple une table de cosinus unique contenant 128 échantillons. On va voir maintenant comment utiliser cette table de cosinus. On note N_{cos} la longueur de cette table que l'on suppose paire. On va adresser la table comme un buffer circulaire de longueur N_{cos}.

Les valeurs de phase dans la table correspondent à N_{cos} phases Ψ_i uniformément réparties entre – π et π.

$$\Psi_i = \frac{i\pi}{N}, \quad i \in \left[-\frac{N_{cos}}{2}, \frac{N_{cos}}{2} - 1 \right]$$

On note deb_cos l'adresse du début de cette table, pointant sur la valeur du cosinus de l'angle – π. On note mid_cos l'adresse du milieu de la table. La valeur cos (Ψ_i) est rangée à l'adresse mid_cos + *i*. On appelle *i* l'index de la table.

La variable phase a été calculée dans les équations précédentes sans souci de la maintenir entre – π et π, aussi faut-il la convertir modulo 2π pour la ramener dans l'intervalle [– π, π[correspondant aux angles de la table et pour éviter que les valeurs ne deviennent trop grandes. Il faut comparer phase à – π et + π et quand phase est plus grand que π lui enlever 2π, ou quand phase est inférieur à – π lui ajouter 2π. Il est possible, dans certains cas, de supprimer ces tests coûteux en temps. Il suffit d'adresser la table des cosinus comme un buffer circulaire, on en verra un exemple pour la génération des porteuses.

On va voir maintenant les opérations pour la méthode de calcul de la variable phase entre – π et π, de son cosinus et de son sinus.

Comme la normalisation de la phase entre – π et + π est faite assez souvent, on a écrit une macro pour le faire. Cette macro s'appelle testa02pi.

On suppose que la variable phase est ramenée entre – π et π. La valeur de la phase (modulo 2π) dont on cherche le cosinus et représentée dans le DSP sur 16 bits par un nombre entier noté I_{phase}. La valeur minimum de phase – π est codée par -2^{15} et la valeur maximale $\pi(1 - 2^{-15})$ par $2^{15} - 1$. L'entier I_{phase} qui est lié à phase par :

$$I_{phase} = \frac{2^{15}\, phase}{\pi}$$

Dans le programme, on travaillera avec l'entier.

L'incrément de phase entre 2 valeurs de la table vaut $2\pi/N_{cos}$ et correspond à un incrément de I_{phase} égal à $2^{16}/N_{cos}$. Pour un entier I_{phase} donné (c'est-à-dire une

valeur phase donnée), l'index i de la table est égal à $i = N_{cos} I_{phase} 2^{-16}$. En général, N_{cos} est une puissance de 2 et i est obtenu par un simple décalage de I_{phase}. Avec les valeurs choisies pour l'exemple :

$$N_{cos} 2^{-16} = 2^7 2^{-16} = 2^{-9}$$

Il faut donc décaler I_{phase} de 9 bits à droite pour obtenir l'index de lecture dans la table de cosinus.

On a vu comment générer un cosinus à partir d'une table de cosinus. Si on veut utiliser la même table pour générer le sinus d'un angle α, il suffit de calculer le cosinus d'un angle complémentaire $\pi/2 - \alpha$, ou $-\pi/2 + \alpha$, $3\pi/2 + \alpha$, c'est-à-dire si i est l'index de lecture du cosinus, d'utiliser l'index $(i - N_{cos}/4)$, si i est supérieur ou égal à $-N_{cos}/4$, ou l'index $i + 3N_{cos}/4$ sinon.

Le même type de méthode va être utilisé pour générer les 2 porteuses en quadrature par lecture en table.

Génération de 2 porteuses en quadrature

On va voir comment générer une porteuse $\cos(2\pi f_c t)$, la génération de la porteuse $\sin(2\alpha f_c t)$ s'en déduisant par la remarque précédente sur l'utilisation de l'angle complémentaire.

Dans l'application de modulation GMSK, les porteuses ne sont en général pas générées par le DSP mais par des oscillateurs analogiques car f_c est souvent une fréquence élevée. Les 2 signaux $x_I(t) = \cos(\Phi(t))$ et $x_Q(t) = \sin(\Phi(t))$ sont générés par le DSP, puis convertis par des convertisseurs analogiques/numériques et multipliés par des multiplieurs analogiques avec les 2 porteuses en quadrature.

On étudie cependant la génération de porteuses de fréquence f_c assez basse.

On note $p(t) = \cos(2\pi f_c t) = \cos(\alpha(t))$, la porteuse. Pour créer ce signal, on va calculer l'argument $\alpha(t)$ du cosinus, puis lire en table la valeur du cosinus de cet angle. Le signal est échantillonné à la fréquence f_e. On note T_e la période d'échantillonnage. La fréquence f_e est au moins 2 fois plus grande que f_c pour respecter le théorème de Shannon. On peut écrire :

$$\alpha(nT_e) = 2\pi f_c nT_e = \alpha((n-1)T_e) + 2\pi f_c T_e = \alpha((n-1)T_e) + \Delta\alpha$$

La phase $\alpha(nT_e)$ peut donc se calculer récursivement en ajoutant à la phase précédente l'incrément de phase $\Delta\alpha$. On obtient ensuite son cosinus par lecture en table.

La précision de la valeur de la fréquence générée dépend de la précision avec laquelle on peut représenter cet incrément $\Delta\alpha$ sur le DSP. Cet incrément de phase correspond à un incrément de ΔI_α sur l'entier qui représente α. Cet incrément

ΔI_α est égal à $2^{16} f_c T_e$ arrondi à l'entier le plus proche. Si le nombre d'échantillons par période est un nombre entier puissance de 2 (2^k par exemple), alors $\Delta I_\alpha = 2^{16} f_c T_e = 2^{(16-k)}$ peut être représenté exactement et la précision de la fréquence générée ne dépend plus que de la précision sur la fréquence d'échantillonnage. Si ce n'est pas le cas, on introduit une erreur df_c sur f_c qui est bornée par :

$$|df_c| < 2^{-17} f_c$$

Par ailleurs, comme les cosinus sont obtenus par lecture en table et que la table est de longueur finie, on introduit une erreur sur les amplitudes de la porteuse quand la phase α est arrondie à la phase en table Ψ_i la plus proche. On peut améliorer la précision par interpolation entre valeurs Ψ_i voisines, mais on ne le fera pas dans cet exemple.

Implantation sur un C54x de la génération de 2 porteuses en quadrature

On va d'abord tester la génération de porteuse, puis on testera la génération des signaux x_I et x_Q. On suppose que $f_e = 1/T$ où T est la période bit, et que $f_e/f_c = 8 = Ns$. La table de cosinus est formée de 128 valeurs de cosinus en format Q_{14}. Elle est pointée par deb_cos. On suppose que la table de cosinus est initialisée dans une section nommée tab_cos. L'incrément ΔI_α vaut ici 2^{13} car $2^{16} f_c T_e = 2^{(16-3)}$.

On est ici dans le cas simple où f_e/f_c est entier (égal à 8). La table de cosinus ayant 128 points, il suffit de lire cette table en faisant des sauts d'index égaux à 16 ($2^7/2^3$) pour générer une sinusoïde avec 8 échantillons par période. C'est la méthode que nous utiliserons ici. On peut travailler directement sur l'index i, on n'a pas à passer par I.

Pour générer la porteuse cosinus, on utilise la table de cosinus et on l'adresse comme un buffer circulaire de longueur N avec $AR1$ comme pointeur. Le registre $AR1$ est initialisé avec l'adresse de début du buffer deb_cos, et on incrémente ce pointeur du contenu de $AR0$ égal à 16.

Pour générer la porteuse sinus, on utilise le même buffer circulaire, mais on l'adresse avec $AR2$. On initialise $AR2$ à deb_cos + $N_{cos}/4$ et on décrémente ensuite $AR2$ de $AR0 = 16$.

Le *listing 8.17* donne le programme C54x de la génération de 2 porteuses en quadrature. On appelle porteuse.asm le fichier assembleur correspondant.

Le programme étant testé avec le simulateur *Code Composer*, on associera un fichier de sortie à chacune des porteuses, l'écriture dans le fichier se faisant lorsqu'un échantillon est écrit dans le registre de transmission du port série 0 $DXR0$.

Dans cet exemple, la table de cosinus va de l'angle 0 à l'angle 2π.

Listing 8.17 – Génération de 2 signaux de fréquence pure en quadrature (porteuses en quadrature).

```
                .mmregs
                .global     début,boucle
Ncos            .set        128
Nsur2           .set        64
Nsur4           .set        32
Inc             .set        16
* Table de cosinus
          .sect "tab_cos"
deb_cos   .word 16384,16364,16305,16207,16069,15893,15679,15426
          .word 15137,14811,14449,14053,13623,13160,12665,12140
          .word 11585,11003,10394,9760,9102,8423,7723,7005
          .word 6270,5520,4756,3981,3196,2404,1606,804
          .word 0,-804,-1606,-2404,-3196,-3981,-4756,-5520
          .word -6270,-7005,-7723,-8423,-9102,-9760,-10394,-11003
          .word -11585,-12140,-12665,-13160,-13623,-14053,-14449,-14811
          .word -15137,-15426,-15679,-15893,-16069,-16207,-16305,-16364
          .word -16384,-16364,-16305,-16207,-16069,-15893,-15679,-15426
          .word -15137,-14811,-14449,-14053,-13623,-13160,-12665,-12140
          .word -11585,-11003,-10394,-9760,-9102,-8423,-7723,-7005
          .word -6270,-5520,-4756,-3981,-3196,-2404,-1606,-804
          .word 0,804,1606,2404,3196,3981,4756,5520
          .word 6270,7005,7723,8423,9102,9760,10394,11003
          .word 11585,12140,12665,13160,13623,14053,14449,14811
          .word 15137,15426,15679,15893,16069,16207,16305,16364

.text
* Initialisations de  DP à 0, et de la phase Ialpha à 0
* La phase Ialpha est dans l'accu A, et l'indice en table dans B*
* attention on travaille dans la partie haute des accus
* On initialise AR1 à mid_cos et AR0 à Nsur4
debut :
          LD    #0, DP
          LD    #0, A
          STM   #Ncos,BK
          STM   #deb_cos,AR1
          STM   #(deb_cos+Nsur4),AR2
          STM   #Inc, AR0

* Boucle sans fin
boucle :
          LD    *AR1+0%,A
          STL   A, DXR0
          LD    *AR2-0%,B
          STL   B,DXR0
* Retour au début de la boucle sans fin
     B    boucle
```


Dans le fichier de commande du linker, on déclare et on alloue la section tab_cos en mémoire données et en l'alignant sur une adresse multiple de 256 pour que le buffer circulaire fonctionne bien.

La *figure 8.27* illustre le test de la génération de 2 porteuses en quadrature (déphasée de $\pi/2$).

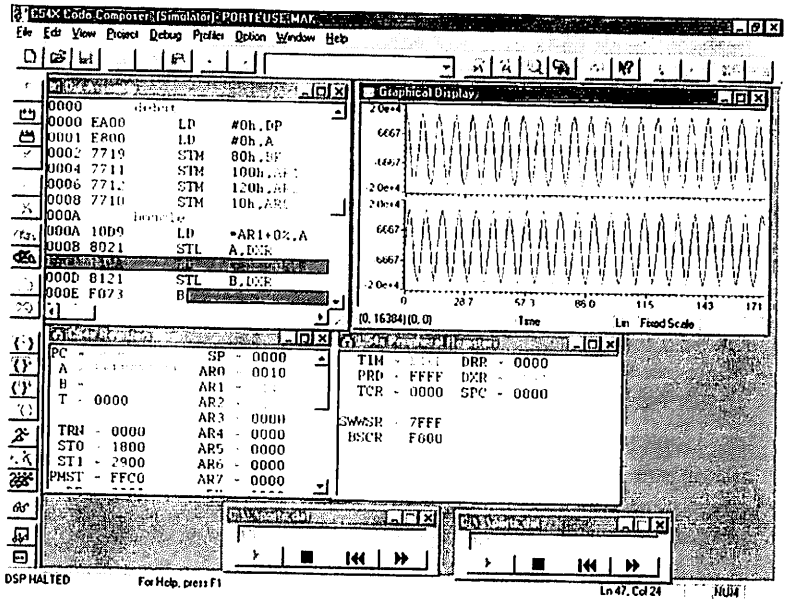

Figure 8.27 – Test de la génération de 2 porteuses en quadrature.

Le *listing 8.18* est le fichier de commande de l'éditeur de liens pour le programme porteuse.asm.

Listing 8.18 – Fichier de commandes de l'éditeur de liens pour le programme porteuse.asm.

```
-m porteuse.map
-e debut
MEMORY  /* TMS320C54x microprocessor mode memory map */
{
      PAGE 0 :
        PROG :    origin = 0x0000, length = 0x10000
      PAGE 1 :
```

```
      DATA :     origin = 0x0060, length = 0xFFA0
}
SECTIONS
{
      .text        : load = PROG          page 0
      .data        : load = DATA          page 1
      .bss         : load = DATA          page 1
      tab_cos align(256)   load = DATA    page 1
}
```

Implantation sur C54x de la modulation GMSK

On va tester la modulation GMSK sur une suite binaire périodique avec 10 bits par période. Cette suite est la suivante :

$an = |0\ 1\ 1\ 0\ 0\ 1\ 0\ 1\ 0\ 0|$

On range ces bits dans un buffer circulaire de taille NB = 10 en déclarant une section bits. On alloue la section dans le fichier de commandes en l'alignant sur une adresse multiple de 16. On utilise AR5 comme pointeur. L'adresse de début de la table est deb_bit.

On va d'abord générer le signal de phase $\Phi(t)$, puis les 2 signaux x_I et x_Q. On utilise une fréquence d'échantillonnage f_e égale à 8 fois la fréquence bit $1/T$. On appelle *NS* le nombre d'échantillons par bit, ici NS = 8.

On utilise la méthode présentée précédemment pour calculer la phase (variable phase) puis le cosinus et le sinus de cette phase pour générer x_Q et x_I. On teste le programme avec *Code Composer* en 2 étapes. Dans la première on teste la phase et dans la deuxième les signaux x_I et x_Q.

Dans la première étape, on sauve la phase en mémoire au fur et à mesure dans un buffer pointé par AR1. L'adresse de départ de ce buffer est résu. Le nombre maximum d'échantillons que l'on peut sauvegarder dépend de la taille de la section bss qui réserve le buffer resu.

Dans la deuxième étape, on ne sauve plus la phase. On calcule le cosinus et le sinus de la phase, c'est-à-dire x_I et x_Q. On écrit x_I dans le registre DXR0 et x_Q dans le registre DXR1, et on associe un fichier de sortie à chacun de ces registres.

Génération de la phase Φ(t)

Le *listing 8.19* est le programme de génération de la phase $\Phi(t)$. On appelle phase.asm le fichier assembleur correspondant. Dans ce listing on n'a pas inséré la section tab_cos. Elle est dans un fichier appelé tabcos.asm que l'on compile et que l'on lie avec le fichier xixq.asm. Il en est de même pour la table contenant les échantillons de la partie évolutive de $\varphi(t)$, à savoir les tableaux *phi*03 ou *phi*05 donnés plus haut. Dans le listing, on a pris l'exemple BT = 0,3, c'est-à-dire qu'on utilise le tableau *phi*03 (et donc le fichier *phi*03.asm). Le tableau *phi*03 est pointé

par AR4 comme un buffer circulaire de taille *Nphi*. Son adresse de début est notée *phi*. Dans le fichier de commande de l'éditeur de liens, on alloue ce tableau en alignant sur une adresse multiple de 64.

D'autre part, on réserve à l'aide d'une section.usect un tableau de *Nphi* points où *Nphi* est la longueur de la partie variable de $\varphi(t)$. On utilise ce tableau comme un buffer circulaire de longueur *Nphi*. L'adresse de début de ce tableau est deb_phase et son pointeur est AR3.

Listing 8.19 – Génération de la phase du signal GMSK.

```
            .mmregs
            .global     debut,boucle
            .global     deb_cos, phi
            .global     deb_phase
            .global     resu
Nphi        .set        32
Nphim       .set        -32
L           .set        4
Ncos        .set        128
Nsur2       .set        64
Nsur4       .set        32
NB          .set        10
NS          .set        8
NSm         .set        -8+Nphi
unmL        .set        1-L


            .sect       "bits"
deb_bit     .word       -1,1, 1, -1, -1, 1, -1, 1, -1, -1
deb_phase   .usect      "phase",Nphi
            .bss        phimem,1
            .bss        resu,1000
* Début de la macro pour ramener la phase entre -π et π
testa02pi   .macro
            SUB   #8000h,A,B        ; on soustrait -2^(15)
            BC    suitel ?, BGEQ    ; test si phase entre 0 et 2pi
            SUB   #8000h,A          ; on ajoute 2^(15)
            SUB   #8000h,A          ; on ajoute 2^(15)
            B suite ?
suitel ?    ADD   #8000h,A,B        ; on ajoute -2^(15)
            BC    suite ?,BLT
            ADD   #8000h,A          ; on soustrait 2^(15)
            ADD   #8000h,A          ; on soustrait 2^(15)
suite ?           ; fin test phase entre 0 et 2pi
            .endm

            .text
```

```
* Initialisation de BK, DP à 0, accus à 0, et les ARx
début :     LD          #0, DP
            LD          #0, A
            LD          #0, B
            STM         #deb_bit,AR5
            STM         #deb_cos,AR2
            STM         #deb_phase,AR3
            STM         #phi,AR4
            STM         #resu,AR1
            STM         +1,AR0
            STM         #Nphi, BK
            STM         #(NS-1),AR7
            RSBX        OVM
            SSBX        SXM
* Initialisation du buffer de phase à 0
            RPT         #(Nphi-1)
            STL         A,*AR3+%
*initialisation de phimem
            LD          *AR5,T
            MPY         #04000h,A       ; -2^(14) an(1)(pi/2 an(1))
            STL         A,*(phimem)

* traitement des L premiers bits
            STM         #(L-1),AR6
Ldeb        STM         #Nphi-1,BRC
            RPTB        fin-1           ; de k=0 à Nphi
            LD          *AR3,A          ; accu=phase(k)
            MAC         *AR4+0%,*AR5,A  ; A=phase(k)+an(i) phi(k)
            testa02pi
            STL         A,*AR3+%        ; A->phase(k)
            NOP
Fin         NOP
            STM         #(NS-1), AR7
nsbouc      LD          *AR3,A          ; on sort NS valeurs
            STL A,      *AR1+           ; et on remet à 0 NS cases
            ST          #0,*AR3+%
            BANZ        nsbouc,*AR7-

            STM         #NB, BK
            MAR         *AR5+%
            STM         #Nphi,BK
            BANZ        Ldeb, *AR6-
* traitement des bits suivants,
* début de la boucle infinie
boucle      STM         #Nphi-1,BRC
            RPTB        fin2-1          ; de k=0 à Nphi
            LD          *AR3,A          ; accu=phase(k)
            MAC         *AR4+0%,*AR5,A  ; A=phase(k)+an(i) phi(k)
            testa02pi
```



```
            STL         A,*AR3+%        ; A->phase(k)
fin2
*On ajoute phimem aux NS premiers points du tableau puis on les sort
* et on remet à 0 la case dans phase
            STM         #NS-1,AR7
nsbouc2     LD          *(phimem),B
            ADD         *AR3,B,A        ; B=phimem+phase(k)
            testa02pi
            STL         A,*AR1+
            ST          #0,*AR3+%       ; 0->phase(k)
            BANZ        nsbouc2,*AR7-
fin3
* actualisation de phimem
            LD          *(phimem),A
            STM         #NB, BK
            MAR         *+AR5(#unmL)%
            LD          *AR5,T
            MAC         #04000h,A       ; -2^(14) an(1)(pi/2 an(1))
            testa02pi
            STL         A,*(phimem)
            MAR         *+AR5(#L)%
            STM         #Nphi,BK
            B           boucle
.end
```

Le *listing 8.20* représente le fichier tabcos contenant la table de cosinus en Q_{14}.

Listing 8.20 – Fichier tabcos.asm, table de cosinus de – π à π en Q_{14}.

```
            .def deb_cos,mid_cos
            .sect "tab_cos"
deb_cos     .word -16384,-16364,-16305,-16207,-16069,-15893,-15679,-15426
            .word -15137,-14811,-14449,-14053,-13623,-13160,-12665,-12140
            .word -11585,-11003,-10394,-9760,-9102,-8423,-7723,-7005
            .word -6270,-5520,-4756,-3981,-3196,-2404,-1606,-804
            .word 0,804,1606,2404,3196,3981,4756,5520
            .word 6270,7005,7723,8423,9102,9760,10394,11003
            .word 11585,12140,12665,13160,13623,14053,14449,14811
            .word 15137,15426,15679,15893,16069,16207,16305,16364
            .word 16384,16364,16305,16207,16069,15893,15679,15426
            .word 15137,14811,14449,14053,13623,13160,12665,12140
            .word 11585,11003,10394,9760,9102,8423,7723,7005
            .word 6270,5520,4756,3981,3196,2404,1606,804
            .word 0,-804,-1606,-2404,-3196,-3981,-4756,-5520
            .word -6270,-7005,-7723,-8423,-9102,-9760,-10394,-11003
            .word -11585,-12140,-12665,-13160,-13623,-14053,-14449,-14811
            .word -15137,-15426,-15679,-15893,-16069,-16207,-16305,-16364
mid_cos     .set    deb_cos+64
```

La *figure 8.28* représente la phase $\Phi(t)$ ramenée entre $-\pi$ et π.

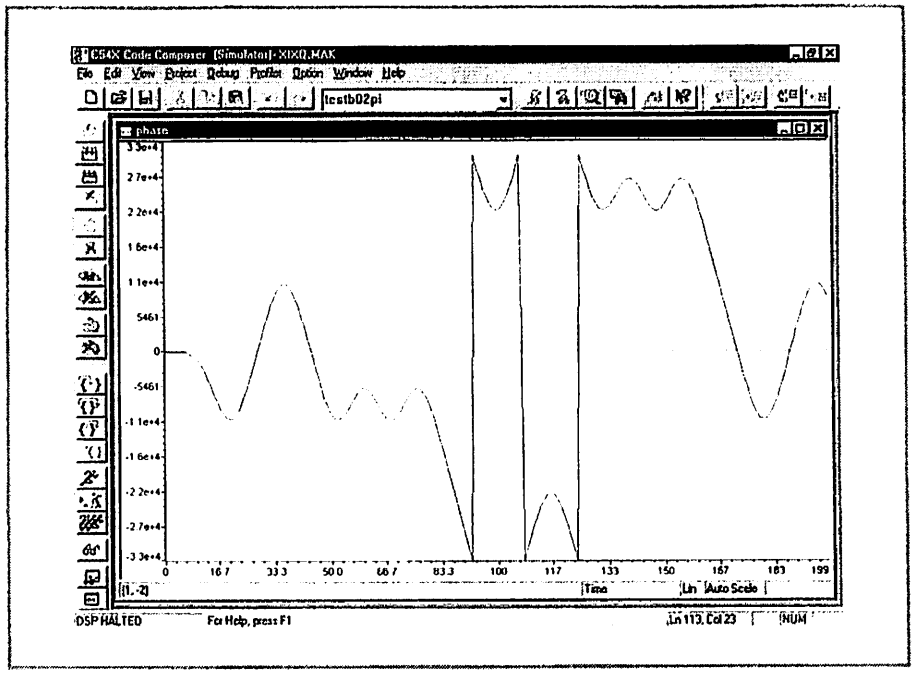

Figure 8.28 – Phase (à un facteur multiplicatif près) du signal modulé GMSK ramenée entre $-\pi$ et π.

Génération des signaux x_I et x_Q

On va maintenant générer les signaux x_I et x_Q, et les sortir sur les registres de ports série *DXR*0 et *DXR*1 pour le test avec *Code Composer*. On ne sauvegarde plus la phase.

Le *listing 8.21* est le programme de génération des signaux x_I et x_Q. Le fichier s'appelle xixqcos.asm.

Listing 8.21 – Génération des signaux x_I et x_q de la modulation GMSK.

```
            .mmregs
            .global     debut,boucle
            .global     deb_cos, mid_cos,phi
            .global     deb_phase
            .global     resu
Nphi        .set        32
Nphim       .set        -32
```



```
L           .set        4
Ncos        .set        128
Nsur2       .set        64
Nsur4       .set        32
NB          .set        10
Ns          .set        8
NSm         .set        -8+Nphi
unmL        .set        1-L

            .sect       "bits"
deb_bit     .word       -1,1, 1, -1, -1, 1, -1, 1, -1, -1

deb_phase   .usect      "phase",Nphi
            .bss        phimem,1
            .bss        resu,1000
* macro pour ramener la phase entre 0 et 2 π
testa02pi   .macro
            SUB         #8000h,A,B      ; on soustrait -2^(15)
            BC          suitel ?, BGEQ  ; test si phase entre 0 et 2pi
            SUB         #8000h,A        ; on ajoute 2^(15)
            SUB         #8000h,A        ; on ajoute 2^(15)
            B suite ?
suitel ?    ADD         #8000h,A,B      ; on ajoute -2^(15)
            BC          suite ?,BLT
            ADD         #8000h,A        ; on soustrait 2^(15)
            ADD         #8000h,A        ; on soustrait 2^(15)
suite ?                                 ; fin test phase entre 0 et 2pi
            .endm

            .text
* Initialisation de BK, DP à 0,SXM, OVM, accus à 0, et les ARx
début :     LD          #0, DP
            LD          #0, A
            LD          #0, B
            STM         #deb_bit,AR5
            STM         #deb_cos,AR2
            STM         #deb_phase,AR3
            STM         #phi,AR4
            STM         #resu,AR1
            STM         +1,AR0
            STM         #Nphi, BK
            STM         #(NS-1),AR7
            RSBX        OVM
            SSBX        SXM
* Initialisation du buffer de phase à 0
            RPT         #(Nphi-1)
            STL         A,*AR3+%
*initialisation de phimem
            LD          *AR5,T
```

```
            MPY         #04000h,A       ; -2^(14) an(1)(pi/2 an(1))
            STL         A,*(phimem)

* traitement des L premiers bits
            STM         #(L-1),AR6
Ldeb        STM         #Nphi-1,BRC
            RPTB        fin-1           ; de k=0 à Nphi
            LD          *AR3,A          ; accu=phase(k)
            MAC         *AR4+0%,*AR5,A  ; A=phase(k)+an(i) phi(k)
            testa02pi
            STL         A,*AR3+%        ; A->phase(k)
fin                     NOP
            STM         #(NS-1), AR7
nsbouc                  LD *AR3,A       ; on sort NS valeurs
            SFTA        A,-9,A
            ADD         #mid_cos,A,B
            STLM        B,AR1
            NOP
            NOP
            LD          *AR1,B
            STL         B,DXR0

            BC          sui,BGT
            ADD         #(Nsur2),B
            B           suil
Sui         SUB         #Nsur2,B
suil        ADD         #mid_cos,B

            STLM        B,AR1
            NOP
            NOP
            LD          *AR1,A
            STL         A,DXR1
            ST          #0,*AR3+%
            BANZ        nsbouc,*AR7-

            STM         #NB, BK
            MAR         *AR5+%
            STM         #Nphi,BK
            BANZ        Ldeb, *AR6-
* traitement des bits suivants,
* début de la boucle infinie
boucle      STM         #Nphi-1,BRC
            RPTB        fin2-1          ; de k=0 à Nphi
            LD          *AR3,A          ; accu=phase(k)
            MAC         *AR4+0%,*AR5,A  ; A=phase(k)+an(i) phi(k)
            testa02pi
            STL         A,*AR3+%        ; A->phase(k)
fin2
```



```
* On ajoute phimem aux NS premiers points du tableau puis on les sort
* et on remet à 0 la case dans phase
            STM         #NS-1,AR7
nsbouc2     LD          *(phimem),B
            ADD         *AR3,B,A        ; A=phimem+phase(k)
            testa02pi
            SFTA        A,-9,A
            ADD         #mid_cos,A,B
            STLM        B,AR1
            NOP
            NOP
            LD          *AR1,B
            STL         B,DXR0
* Calcul du sinus
            NOP
            ADD         #Nsur4,A,B
            NOP
            NOP
            BC          suii,BGT
            ADD         #(Nsur2),B
            B           suii1
suii        SUB         #(Nsur2),B
suii1       ADD         #mid_cos,B
            STLM        B,AR1
            NOP
            NOP
            LD          *AR1,A
            STL         A,DXR1
            ST          #0,*AR3+%       ; 0->phase(k)
            BANZ        nsbouc2,*AR7-
fin3
* actualisation de phimem
            LD          *(phimem),A
            STM         #NB, BK
            MAR         *+AR5(#unmL)%
            LD          *AR5,T
            MAC         #04000h,A       ; -2^(14) an(1)(pi/2 an(1))
            testa02pi
            STL         A,*(phimem)
            MAR         *+AR5(#L)%
            STM         #Nphi,BK
            B           boucle
            .end
```

Dans ce listing, on considère que l'on travaille avec $BT = 0{,}3$. On doit donc faire l'édition de lien avec le fichier phi03.asm qui contient la partie évolutive de $\varphi(t)$ pour $BT = 0{,}3$. Dans ce fichier la phase $\varphi(t)$ a été multiplié par $2^{15}/\pi$ et arrondie à l'entier le plus proche. Le fichier contient 32 échantillons car $\varphi(t)$ dure 4 bits et qu'il y a 8 échantillons par bit.

Le *listing 8.22* donne le listing phi03.asm.

Listing 8.22 – Listing du fichier phi03.asm.

```
.ref phi
      .sect "phi"
phi   .word 1,2,5,13,29,65,133,256
      .word 465,795,1284,1965,2858,3961,5252,6685
      .word 8192,9699,11132,12423,13526,14419,15100,15589
      .word 15919,16128,16251,16319,16355,16371,16379,16382
```

La *figure 8.29* illustre les résultats obtenus par simulation sous *Code Composer* avec $BT = 0{,}3$.

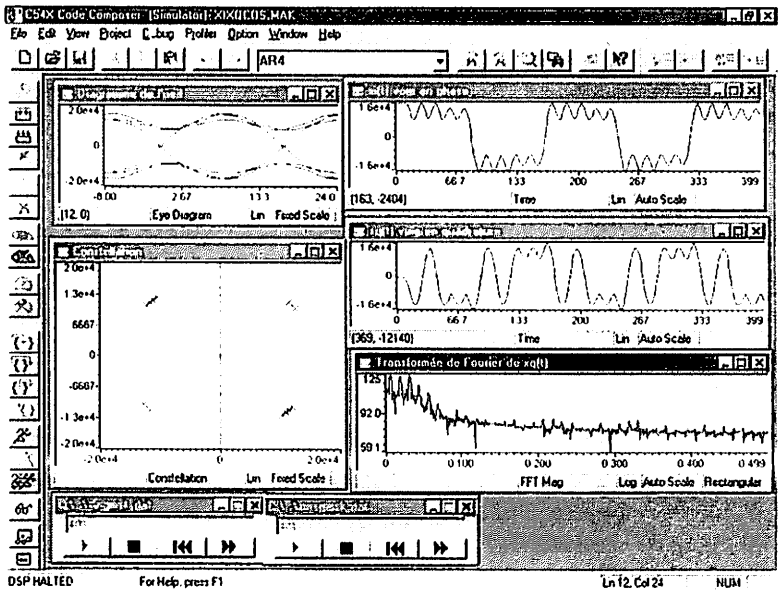

Figure 8.29 – Modulation GMSK, signaux en bande de base x_i et x_q pour BT = 0,3.

On a utilisé pour cette figure plusieurs des fonctionnalités du simulateur *Code Composer*. On a tracé les signaux x_I et x_Q en chargeant en mémoire les fichiers où ils avaient été sauvegardés. La transformée de Fourier de ces signaux donne une idée de l'occupation spectrale de la modulation GMSK. On a tracé le diagramme de l'œil de $x_Q(t)$. On peut constater sur ce diagramme que l'interférence inter-

symbole est assez grande pour $BT = 0{,}3$. Enfin, on a construit la constellation. La constellation s'obtient en traçant xQ en fonction de xI et en prenant un seul échantillon par symbole (1 point sur 8).

Pour comparer l'influence du produit BT sur les signaux x_I et x_Q, on modifie le programme pour passer à $BT = 0{,}5$. Il suffit de changer les valeurs de *Nphi*, de *Nphim* et de *L* en les passant à 24, – 24 et 3. On utilise le fichier phi05.asm contenant 24 échantillons. *Le listing 8.23* représente phi05.asm.

Listing 8.23 – Listing du fichier phi05.asm.

```
      .ref phi
      .sect "phi"
phi   .word 0,1,3,13,50,154,402,897
      .word 1732,2945,4495,6285,8192,10099,11889,13439
      .word 14652,15487,15982,16230,16334,16371,16381,16383
```

La *figure 8.30* illustre les résultats obtenus par simulation sous *Code Composer* avec $BT = 0{,}5$.

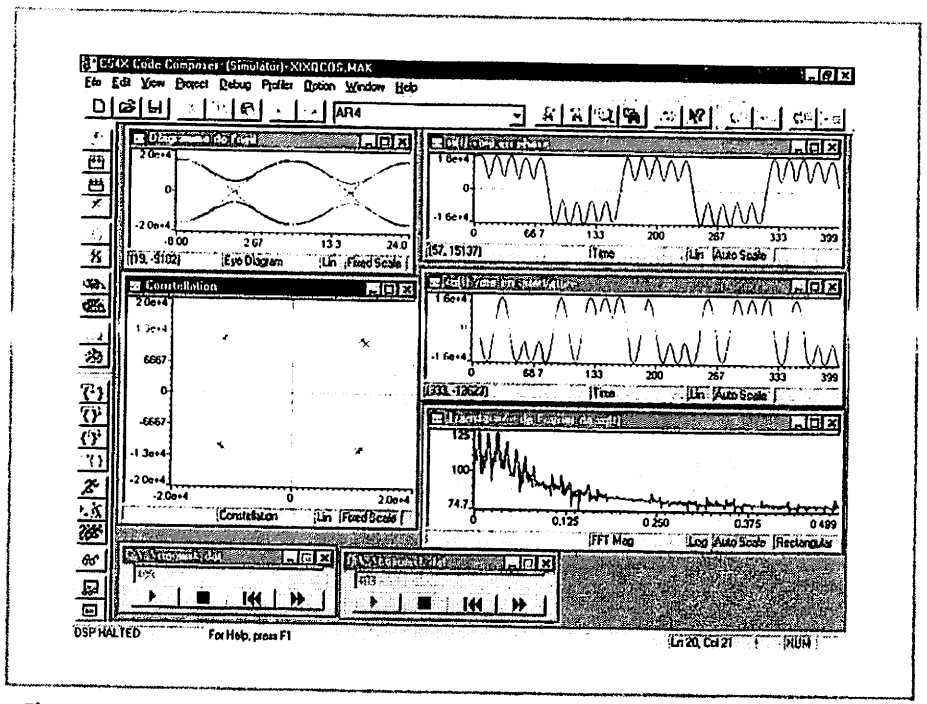

Figure 8.30 – Modulation GMSK, signaux en bande de base x_i et x_q pour BT = 0,5.

On constate, sur le diagramme de l'œil ou la constellation, qu'il y a moins d'interférences intersymbole pour $BT = 0{,}5$. Par contre, la transformée de Fourier montre que le spectre est plus large.

Vecteur d'interruption pour le reset, initialisation de la pile

Dans une application en temps réel, il faudrait ajouter à ce programme un vecteur d'interruption pour le *Reset*. Au démarrage du processeur, l'adresse FF80 est automatiquement chargée dans le compteur programme. Les adresses d'interruption sont espacées de 4 mots. Ainsi, on peut stocker dans ces 4 mots par exemple une instruction de branchement retardée suivie de 2 instructions d'un mot ou d'une instruction de 1 mot. Par exemple, à l'adresse FF80, on peut ranger une instruction de branchement au début du programme. En général, on stocke ces vecteurs d'interruptions dans une section nommée, que l'on affectera dans le fichier de commandes de l'éditeur de liens à l'adresse FF80.

Le *listing 8.24* donne un exemple de fichier contenant un vecteur d'interruption pour le *Reset*.

Listing 8.24 – Listing du fichier vectors.asm, vecteur d'interruption pour le reset.

```
      .mmregs
      .ref debut
      .sect "vectors"
      B debut
```

Avec ce fichier, au moment du *Reset*, le DSP effectue un branchement à l'adresse debut.

Le *listing 8.25* donne un fichier de commandes pour l'éditeur de liens affectant la section vectors à l'adresse FF80.

Listing 8.25 – Fichier de commande de l'éditeur de lien allouant la section vectors à l'adresse ff80.

```
MEMORY/* TMS320C54x microprocessor mode memory map */
{
      PAGE 0 :
        PROG :    origin = 0x00800  , length = 0x100
        VECS :    origin = 0xFF80   , length = 0x20
      PAGE 1 :
        DATA :    origin = 0x0060   , length = 0xFFA0
}

SECTIONS
{
       .text      : load = PROG     page 0
       .data      : load = DATA     page 1
       .bss       : load = DATA     page 1
       .coef      : load = PROG     page 0
       vectors    : load = VECS     Page 0
}
```


Dans une application temps réel, il faut aussi initialiser la pile, pour les interruptions et les appels de subroutines.

Pour réserver de la place pour la pile, on peut utiliser une section.usect avec une longueur suffisante pour l'application. Si deb_pile est l'adresse de début de cette section et TAILLE la longueur de cette section, il faut initialiser le pointeur de pile *SP* avec la valeur deb_pile plus TAILLE. En effet, le pointeur de pile est décrémenté à chaque ajout d'un élément dans la pile. Quand la pile est vide, *SP* doit pointer sur l'adresse la plus élevée de la pile plus 1.

Le *listing 8.26* est un exemple de fichier contenant le vecteur d'interruption pour le *Reset* qui initialise la pile et effectue un branchement au début du programme.

Listing 8.26 – Listing du fichier vectors.asm, vecteur d'interruption pour le *Reset* avec initialisation de la pile.

```
                .mmregs
                .ref debut
TAILLE          .set 200
deb_pile        .usect "PILE",TAILLE
                .sect "vectors"
                BD debut
                STM #STACK+TAILLE,SP
```

Dans le fichier de commande de l'éditeur de liens il faut allouer la section PILE à de la mémoire donnée.

CONCLUSION

Cet ouvrage avait pour objectif de permettre, au travers de différents exemples, d'appréhender les techniques de développement de programmes pour l'implantation d'algorithmes de traitement du signal sur DSP, et plus particulièrement sur les DSP de la famille 320C54x.

Au-delà des exemples présentés, de nombreuses autres applications sont décrites sur le site internet de Texas Instruments (http://www.ti.com/dsps) et plusieurs CD-Rom sont offerts par TI et comportent des notes d'applications ainsi que l'ensemble des documentations techniques. Un des principaux avantages dans l'utilisation des DSP de TI est certainement le grand nombre d'exemples disponibles et accessibles.

D'autres sites dédiés aux processeurs de traitements du signal sont également bien documentés. On trouvera sur le site de Berkeley Design Technology (http://www.bdti.com) les comparaisons des performances (*benchmarks*) entre les DSP de différents constructeurs, ainsi que des analyses sur les nouveaux produits.

Deux autres sites comportent des informations sur les deux grandes conférences du domaine des processeurs de traitement du signal, DSPworld (http://www.dspworld.com) et ICSPAT (même adresse) ainsi que les *proceedings* en accès libre des conférences ICSPAT (1996, 1997 et 1998).

Un autre site d'intérêt plus général dans les domaines de l'électronique et du traitement du signal est celui de Technologie *On Line* (http://www.techonline.com) qui comporte des informations très à jour sur les nouveaux DSP, des tutoriaux ainsi que la possibilité de charger des démonstrations de logiciels de développement.