Architectures pour le calcul de la transformée de Fourier discrète

Architectures for computing the discrete Fourier transform

Jean-Michel MULLER

Laboratoire TIM 3-INPG, 46, avenue F.-Viallet, 38031 GRENOBLE CEDEX.

Jean-Michel Amadeus Muller est ingénieur de l'ENSIMAG et a soutenu en 1985 une thèse de doctorat sur le calcul des fonctions élémentaires. Il est actuellement chargé de recherches au CNRS, et enseigne aux DEA d'informatique et de mathématiques appliquées de l'Université Joseph-Fourier de Grenoble et de l'INPG. Denis Trystram lui a fait partager sa passion de la transformée de Fourier. Son domaine d'intérêt actuel est l'arithmétique des ordinateurs, il est l'auteur d'un livre sur ce sujet.

Denis TRYSTRAM

Laboratoire TIM 3-INPG, 46, avenue F.-Viallet, 38031 GRENOBLE CEDEX.

Denis Trystram est ingénieur de l'ENSIMAG. Il a soutenu un doctorat d'ingénieur à l'INPG en 1984. Il a été assistant à l'École Centrale de Paris pendant 3 années avant d'être nommé Maître de Conférences à l'Université Joseph-Fourier de Grenoble où il enseigne l'Architecture. Il a soutenu une thèse de doctorat à l'INPG en 1988 sur l'analyse de complexité des algorithmes parallèles et systoliques qui constitue son domaine de recherches actuel. Jean-Michel Muller lui a communiqué son intérêt pour la transformation de Fourier et Henri Dutilleux..

RÉSUMÉ

Nous nous intéressons dans cet article aux réalisations matérielles du calcul de la transformée de Fourier discrète. Après une brève présentation du problème, nous passons en revue les principales solutions existantes et nous proposons une architecture spécialisée capable de calculer à la fois en parallèle et en série la transformée de Fourier rapide (TFR) de toute partition arbitraire d'un grand nombre d'échantillons.

MOTS CLÉS

TFR, opérateurs arithmétiques (multiplieurs série), pipe-line, circuits VLSI.

SUMMARY

In this paper, we study a hardware implementation of the Discrete Fourier Transform. After a brief introduction to that problem, we survey the main existing realizations, then we propose a special-purpose architecture able to compute in serial/parallel mode the Fast Fourier Transform (FFT) of arbitrary partitions of N any samples.

KEY WORDS

FFT, Computer Arithmetic (bit serial multiplication), pipeline, VLSI.

1. Introduction

Depuis l'apparition de la technologie des circuits intégrés VLSI (Very Large Scale Integration) [MC80], il est possible d'envisager la résolution rapide d'un nombre croissant de problèmes. Actuellement, les réalisations les plus sophistiquées comportent de l'ordre d'un million de transistors par circuit intégré. La surface d'un circuit est une mesure simple de son coût et de sa complexité, par conséquent, la plupart des choix architecturaux résultent d'un compromis entre la surface et le temps de calcul.

La complexité couplée d'un problème, définie par le produit de la surface par le carré du temps, est connue pour plusieurs applications fondamentales

comme la multiplication [BK81], le tri et la Transformée de Fourier Discrète [THO80].

En ce qui concerne cette dernière application, nombre de résultats théoriques de complexité ont été étudiés [THO83] et conduisent à des solutions asymptotiques que la technologie actuelle ne permet pas d'intégrer. En particulier, il semble difficile actuellement de placer plusieurs dizaines de multiplieurs simples sur un circuit intégré alors que la demande est bien plus importante. Cependant, cette approche est d'un grand intérêt pour permettre la classification et la comparaison des méthodes ainsi que l'aide à l'intuition de solutions et la définition d'outils.

Nous nous intéressons ici aux architectures dédiées à la transformation de Fourier discrète de N échantillons, à l'aide de l'algorithme de la transformée de Fourier rapide (TFR) de Cooley et Tuckey [CT65]. Dans la pratique, N peut atteindre des valeurs bien supérieures à 1 000 et donc, il n'est pas réaliste de proposer des réalisations matérielles sans partitionnement du problème.

Nous proposons une architecture « bit-série » (c'est-à-dire dans laquelle les données sont amenées bit par bit en pipeline [ET77]), comportant seulement huit cellules de base de la complexité d'un multiplieur très simple. Cette solution allie simplicité et haut rendement (elle combine les techniques du calcul parallèle et du calcul série).

2. Quelques rappels sur la Transformation de Fourier

2.1. Intérêt

La Transformation de Fourier Discrète est un outil numérique fondamental que l'on rencontre dans de nombreux domaines des sciences de l'ingénieur, et en particulier dans le domaine du traitement des signaux. Dans beaucoup d'applications elle permet d'obtenir, à partir d'un problème donné, un autre problème plus facile à résoudre (comme passer du domaine en temps au domaine en fréquence, remplacer la convolution par un simple produit, etc.) Elle revêt aussi un aspect théorique important en théorie de la complexité et en calcul formel car elle permet de mettre au point des algorithmes que l'on conjecture optimaux en complexité pour les produits de longs entiers ou de polynômes (algorithme de Strassen, *voir* [AHU 74]). Notons pour finir que la transformation inverse est aussi très utilisée pour la reconstitution des signaux. Ces problèmes ont été largement étudiés.

2.2. Le problème

Étant donné un signal s, discrétisé (*i.e.* dont on connaît la valeur en un nombre fini N de points), il s'agit de calculer les coefficients :

$$c_k = \sum_{n=0,\ N-1} s_n \omega^{nk} \qquad \text{où} \qquad \omega = e^{-2i\pi/N}$$

2.3. Calcul

On peut écrire le problème sous forme matricielle. En effet, posons S égal le vecteur des N échantillons $(s_0, s_1, s_2, \ldots, s_{N-1})$, C le vecteur des solutions $(c_0, c_1, c_2, \ldots, c_{N-1})$ et W la matrice de taille $N \times N$, dont le coefficient courant de la ligne i et de la colonne j est $W_{ij} = \omega^{(i-1)(j-1)}$.

Le problème s'exprime alors de la manière suivante :

Calculer le produit $C = WS$.

Il est bien entendu possible de calculer C en effectuant directement le produit matriciel, c'est l'algorithme de « Transformée de Fourier Directe » (ou DFT) [THO83], mais grâce à la structure très particulière de la matrice W, ce calcul peut s'effectuer plus rapidement par l'algorithme de « Transformée de Fourier Rapide » (TFR). Cet algorithme a été mis au point pour accélérer les calculs sur une machine séquentielle. Cependant, l'algorithme DFT présente l'avantage d'une parallélisation intrinsèque.

2.4. La Transformation de Fourier Rapide (TFR)

La TFR est l'algorithme célèbre, attribué à Cooley et Tuckey [CT65], mais en fait découvert par Runge et Konitz en 1924 [RK24]. Elle est basée sur la remarque suivante :

Le coût du calcul d'une transformée de Fourier discrète de N échantillons (*i.e.* d'un produit matrice vecteur) est de l'ordre de N^2 opérations complexes [$(N-1)^2$ multiplications et $N(N-1)$ additions]. Si N est une puissance de 2, on peut faire beaucoup mieux [passer à une complexité en $O(N \log N)$]. L'idée clé de la TFR est de nature algébrique, elle repose sur des analogies entre les sommes partielles c_k.

En effet, la somme finie c_k s'écrit :

$$c_k = s_0 + s_1 \omega^k + s_2 \omega^{2k} + s_3 \omega^{3k} + \ldots + s_{N-2} \omega^{(N-2)k} + s_{N-1} \omega^{(N-1)k}$$

si l'on regroupe les termes d'ordre pair et impair, on obtient :

$$c_k = P_k + \omega^k I_k$$

où

$$P_k = s_0 + s_2 \omega^{2k} + \ldots + s_{N-2} \omega^{(N-2)k}$$

et

$$I_k = s_1 + s_3 \omega^{2k} + \ldots + s_{N-1} \omega^{(N-2)k}$$

On remarque que $P_k = P_{k+m}$, $I_k = I_{k+m}$ et $\omega^{(k+m)} = -\omega^k$ où $m = N/2$.

D'où l'idée d'économie suivante :

On calcule P_k et $\omega^k I_k$ pour évaluer c_k, et on en déduit sans frais supplémentaires en multiplications la valeur de c_{k+m}. On gagne ainsi un facteur 2 dans le temps du calcul.

De plus, P_k et I_k sont aussi des transformées de Fourier discrètes d'ordre moitié, indépendantes l'une de l'autre. Ceci nous permet de réitérer le procédé sur des intervalles de longueur $m/2$ et ainsi de suite.

Examinons un exemple simple, avec $N = 8$. Nous avons développé le cheminement qui conduit au calcul du coefficient c_3 (attention : il s'agit ici d'un schéma

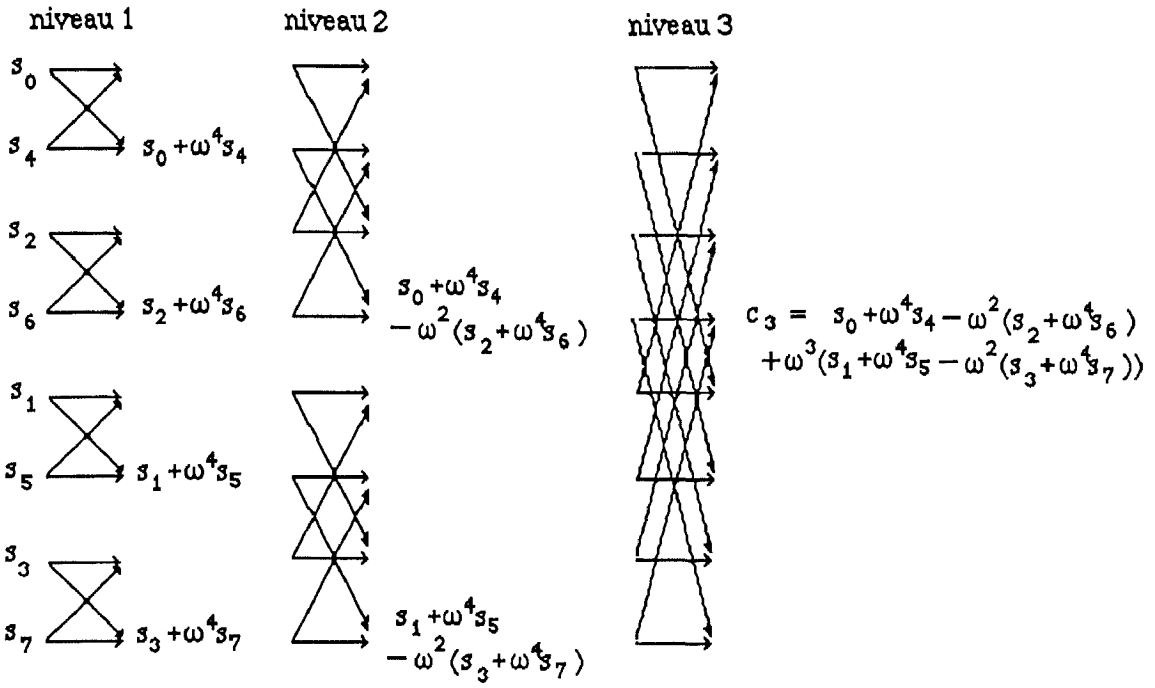
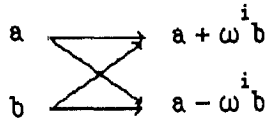


Fig. 1. — Obtention de c_3.

logique, qui ne correspond pas nécessairement à une topologie d'implantation). Le coefficient c_3 est égal à $s_0+s_1\omega^3-s_2\omega^2+s_3\omega+s_4\omega^4-s_5\omega^3+s_6\omega^2-s_7\omega$ (on rappelle que $\omega^4=-1$).

A un niveau donné k, les deux opérations fondamentales à effectuer sont simples. Les puissances i de ω prennent successivement les valeurs $(N/2^k)j$ pour $j=0, 1, \ldots, 2^{k-1}-1$. La figure suivante schématise ces opérations.



C'est la forme de ce petit schéma d'interconnexion qui a donné à l'ensemble du réseau de la figure 1 le nom de « réseau papillon ».

3. Bref survol des différentes architectures

Le problème de la Transformation de Fourier Discrète étant défini, nous allons faire un tour d'horizon sur les propositions de réalisations matérielles existantes pour le résoudre. Nous distinguerons trois approches. D'une part, les propositions concernant les machines générales de type « super calculateurs », puis des architectures à champs d'applications plus réduits, dédiées au domaine du traitement du signal et enfin, les architectures spécialisées uniquement sur ce problème. C'est surtout cette dernière approche qui nous intéresse ici.

3.1. Architectures générales

Le développement récent des supercalculateurs vectoriels ou parallèles a conduit à diverses implantations efficaces de la TFR (comme sur le CRAY-1 [PET83], sur l'IBM 3090-VF [AC86] ou plus récemment sur l'ALLIANT FX-8 [GJ88]). Cependant, ces machines étant très générales, les performances obtenues sont moins bonnes que celles des circuits spécialisés, de plus ces ordinateurs sont beaucoup plus rares parce que très coûteux.

3.2. Quelques architectures dédiées au traitement du signal

Le traitement du signal est un domaine où les algorithmes ont un taux élevé de répétitions d'opérations simples à effectuer en temps réel, avec beaucoup de transferts de données. C'est pourquoi, très tôt, de nombreux concepteurs de circuits s'y sont intéressés (laboratoires Bell, CNET, IBM, NEC, Philips, Texas Instruments, Thomson EFCIS, etc.).

Ainsi, plusieurs réalisations matérielles ont vu le jour (*voir* [REY83] par exemple). Les principes qui président à ces architectures sont simples : au niveau bas, plusieurs unités fonctionnelles indépendantes (jusqu'à des circuits de multiplieurs et additionneurs en cascade pour réaliser un produit scalaire ou des champs d'instructions spécialisés adressables simultanément comme le transfert de données, le calcul, etc.) et à un niveau plus élevé, la gestion simultanée de plusieurs ressources (pipe-line de plusieurs instances...).

3.3. Circuits spécialisés

Dans [THO83], C. D. Thompson donne un tableau comparatif des performances (en complexité) des différents types de circuits spécialisés. Pour une transformée de Fourier discrète de N échantillons, il envisage entre autres le cas de la transformée de Fourier directe (DFT) avec 1, N ou N^2 cellules, ainsi que la

TFR avec une cellule, avec le réseau de la figure 4, avec un réseau d'interconnexion de type « perfect shuffle » (qui correspond à la meilleure manière de mélanger des données, *voir* [STO71]), ou avec une grille carrée de $\sqrt{N} \times \sqrt{N}$ éléments. Voici un extrait de ce tableau, auquel nous avons rajouté la complexité de notre solution. Nous précisons que cette dernière ne vise pas à être optimale en complexité, mais à être implantable efficacement. Il est important de souligner que dès que N devient grand, les circuits optimaux en complexité deviennent irréalistes. Dans l'état actuel de la technologie, le nombre maximal acceptable de cellules élémentaires de TFR que l'on peut placer sur un circuit est de l'ordre de la dizaine.

Méthode	Surface	Temps
DFT 1 cellule	$N \log N$	$N^2 \operatorname{Log} N$
DFT N cellules	$N \log N$	$N \operatorname{Log} N$
DFT N^2 cellules	$N^2 \log N$	$\log N$
TFR 1 cellule	$N \log N$	$N \log^2 N$
Réseau TFR	N^2	$\log N$
TFR « perfect shuffle »	$N^2/\log^2 N$	$\log^2 N$
TFR grille carrée	$N \log^2 N$	$\sqrt{N}$
Notre réseau m cellules	$N+m^2$	$(N \operatorname{Log} N)/m$

La DFT étant par nature très parallèle (il s'agit d'un produit matrice-vecteur), il est possible de l'implanter simplement, de manière systolique sur un réseau orthogonal de N^2 cellules. Mais cette solution devient vite irréaliste à cause de l'encombrement. C'est pourquoi nous chercherons plutôt à implanter la TFR, pour laquelle les solutions directement tirées du schéma de la figure 1 nécessitent $N \log_2 N$ cellules (et même N seulement si l'on organise intelligemment le flot des données). Pour cet algorithme, on peut directement imaginer une architecture qui traduise l'organisation des calculs, à partir d'une cellule élémentaire du type suivant :

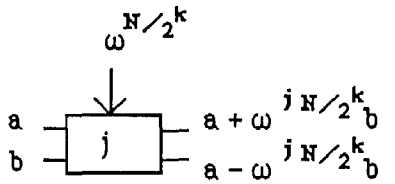


Fig. 2. — Cellule élémentaire.

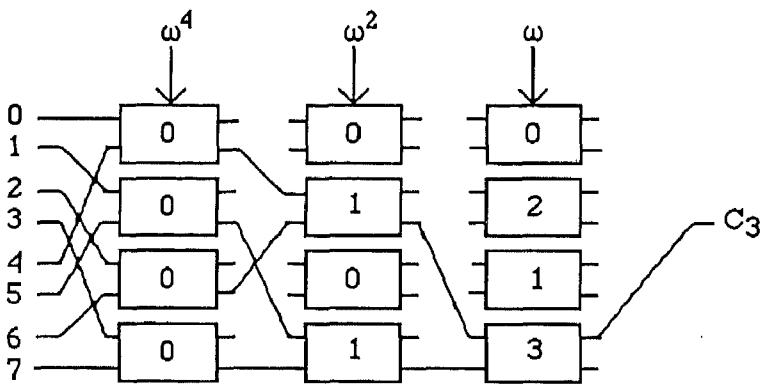


Fig. 3. — Calcul du coefficient c_3 (pour N=8).

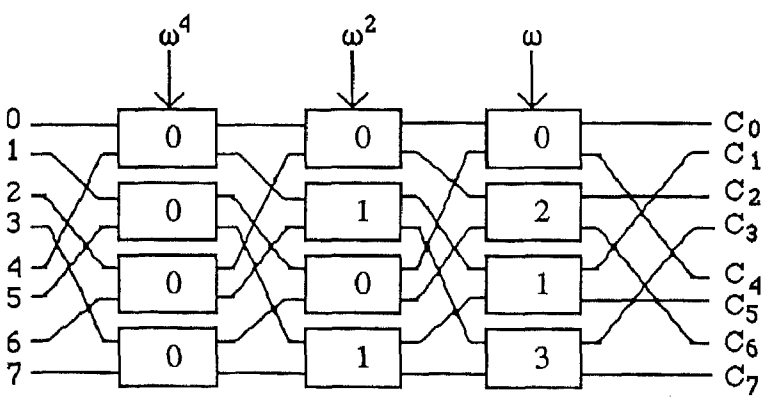


Fig. 4. — Réseau TFR (pour N=8).

Si l'on suit pas à pas l'algorithme précédent pour calculer c_3, on remarque qu'au début s_0 doit rencontrer s_4 pour former $s_0+\omega^4 s_4$. De même, s_2 doit rencontrer s_6 pour former $s_2+\omega^4 s_6$, s_1 doit rencontrer $s_5 (s_1+\omega^4 s_5)$ et s_3 doit rencontrer $s_7 (s_3+\omega^4 s_7)$. A la deuxième étape, le résultat de (s_0, s_4) doit rencontrer celui de (s_2, s_6) pour former $s_0+\omega^4 s_4-\omega^2 (s_2+\omega^4 s_6)$. Enfin, ce résultat doit rencontrer son « symétrique » de la partie impaire : $s_1+\omega^4 s_5-\omega^2 (s_3+\omega^4 s_7)$ de manière à obtenir le coefficient c_3.

Le cheminement est schématisé sur le réseau précédent pour le calcul du coefficient c_3.

On obtient de la même manière le réseau complet.

En généralisant la structure, les coefficients se placent dans un ordre qui suit une permutation miroir. On peut replier les cellules sur elles-mêmes d'un étage à l'autre, l'organisation du flot de données est alors :

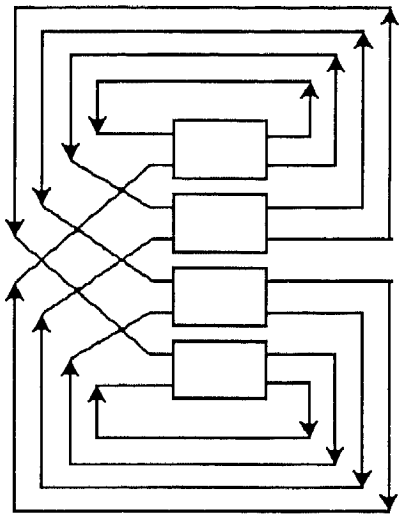


Fig. 5. – Réseau TFR cyclique (pour N = 8).

Pour une meilleure compréhension, étudions sur le même exemple le cheminement correspondant au calcul du coefficient c_3. Il se compose de trois étapes s'exécutant dans le même temps (c'est le temps de calcul de la cellule de base). Nous prendrons ce temps comme unité. Nous représentons ci-dessous ces trois étapes.

4. Multiplication « bit série »

4.1. Situation du problème

Les multiplieurs « bit série » et « série/parallèle », qui fournissent leurs résultats en série, c'est-à-dire bit par bit, sont très intéressants par leur faible encombrement, et ont été largement étudiés [LYO76, CW79]). Ils sont basés sur la considération suivante : si les opérandes que l'on désire multiplier s'écrivent sur un grand nombre de bits, il n'est pas très réaliste de supposer que celles-ci arrivent tous en même temps, « en parallèle » : on supposera ici que les opérandes arrivent *chiffre par chiffre.*

Nous pouvons nous demander s'il n'est pas possible de commencer à calculer *dès l'arrivée des premières données,* sans attendre d'avoir obtenu tous les bits des opérandes. Dans [CW79], Chen et Willoner proposent un multiplieur dont les entrées et les sorties s'effectuent en série, et dont le temps écoulé entre deux sorties successives d'un bit est approximativement le même que pour un additionneur/soustracteur séquentiel simple, ce qui est tout particulièrement intéressant, car on peut alors envisager de pipe-liner l'évaluation d'expressions arithmétiques. Le multiplieur de Chen et Willoner ne permet de traiter que des opérandes positives. Par la suite, plusieurs auteurs ont proposé des solutions permettant de traiter des opérandes écrites en complément à deux [GNA83, RS86]. Il existe également des multiplieurs « parallèle/série », comme celui de Lyon [LYO76] (qui se transforme d'ailleurs très facilement en multiplieur dont les deux opérandes sont acheminées en série).

Supposons donc que nous désirions calculer le produit P de deux entiers positifs A et B codés en base 2 sur n bits comme suit :

$$A = [A_{n-1} A_{n-2} \ldots A_0]$$
$$B = [B_{n-1} B_{n-2} \ldots B_0]$$
$$P = [P_{2n-1} P_{2n-2} \ldots P_0]$$

Adoptons comme unité de temps l'écart entre deux arrivées de chiffres. Dans le multiplieur de Chen-Willoner, au temps i, le multiplieur reçoit A_i et B_i. Il n'est pas difficile de réaliser que pour tout k, P_k ne dépend que des valeurs A_j et B_j d'indice j inférieur ou égal à k, et qu'il dépend de toutes ces valeurs.

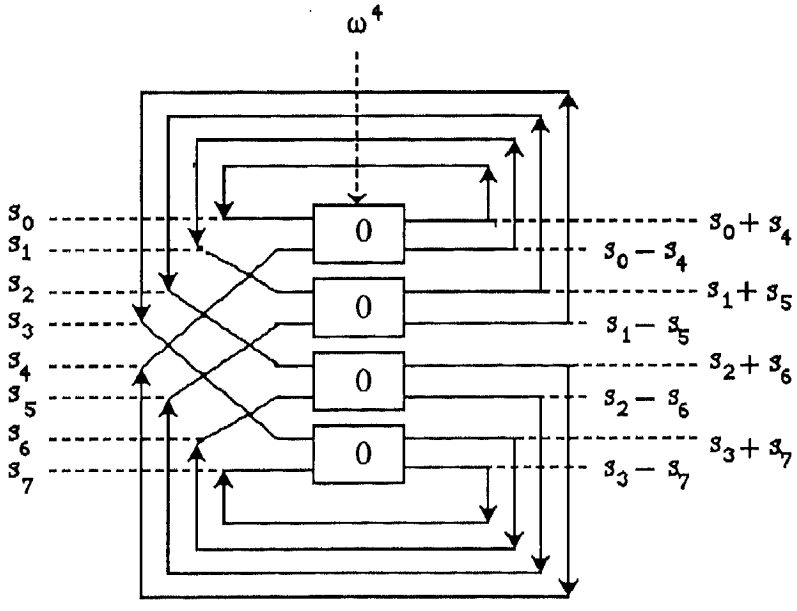


Fig. 6. – Première étape du calcul cyclique (pour N = 8).

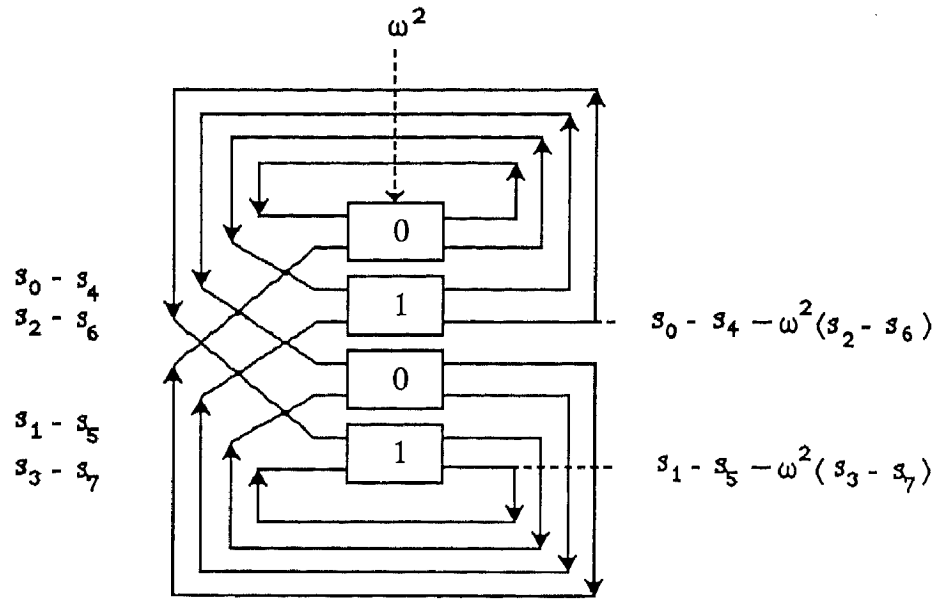


Fig. 7. — Deuxième étape du calcul cyclique de c_3 (pour N = 8).

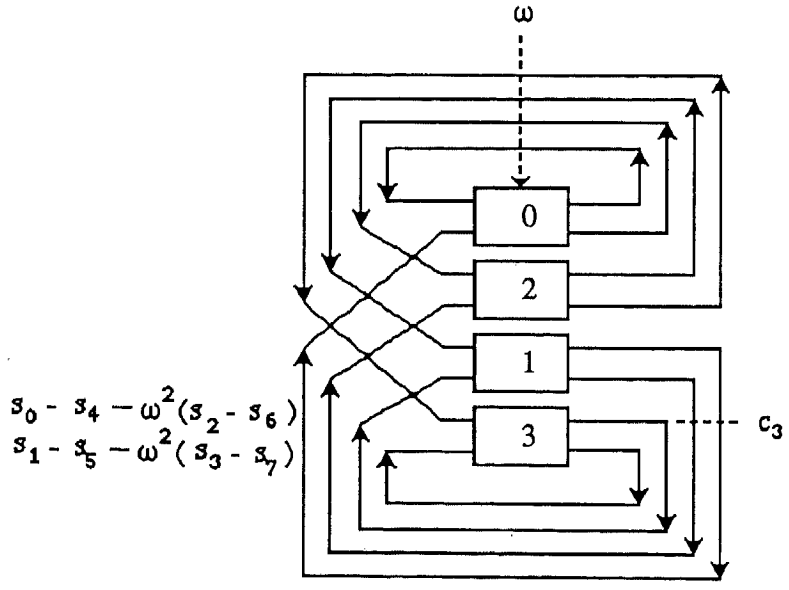


Fig. 8. — Troisième étape du calcul cyclique de c_3 (pour N = 8).

Aussi, le mieux que puisse faire ce multiplieur est de fournir au temps i la valeur de P_{i-1}.

Un tel multiplieur n'est pas très bien adapté au calcul sur des nombres réels écrits en virgule fixe ou en virgule flottante : en effet, seuls les n bits de poids fort de P (P_n, P_{n+1}, P_{n+2}, ..., P_{2n-1}) nous intéressent, et nous devons attendre n unités de temps avant de les obtenir. Le multiplieur de Lyon, lui, grâce à une troncature intelligente des poids faibles, donne en premier P_n, mais seulement n unités de temps après le début du calcul. Un multiplieur où les données entreraient en commençant par les poids forts serait bien plus intéressant pour le problème qui nous concerne. Un tel multiplieur est irréalisable si l'on utilise le système usuel d'écriture des nombres (écriture simple en base 2, ou complément à 2).

En effet, le bit de poids fort du produit P dépend non seulement des bits de poids forts de A et B mais aussi de leurs bits *de poids faibles*, à cause des propagations de retenues. Dans [AVI61], Avizienis a présenté de nouveaux systèmes de notation dits « redondants », ou à « chiffres signés » qui permettent de calculer *sans propagation de retenue*. C'est dans l'un de ces systèmes de notation que notre circuit calculera. Les conversions entre ce système de notation et l'écriture usuelle en base 2 se font sans aucun problème.

Décrivons sommairement le système de notation d'Avizienis que nous utiliserons, appelé système « à chiffres signés » et l'algorithme d'addition (sans propagation de retenue) associé, après quoi nous présenterons notre multiplieur bit-série « poids fort d'abord ».

4.2. La notation a « chiffres signés »

Il s'agit là d'une notation dite *redondante* car elle autorise plusieurs représentations possibles pour certains nombres. Elle utilise des chiffres qui peuvent être *négatifs*. La redondance entraîne un certain « gâchis », car pour représenter un même nombre d'entiers différents il faudra un plus grand nombre de bits qu'avec un autre système de notation; toutefois, cette notation autorise la mise au point d'algorithmes performants pour effectuer certaines opérations arithmétiques, car elle permet d'effectuer des additions exemptes de toute propagation de retenue. Pour la présenter, nous allons nous intéresser à l'écriture en base 2 de nombres entiers avec des chiffres appartenant à $\{-1, 0, 1\}$. Nous conviendrons de noter le chiffre « -1 » : « $\bar{1}$ ». Dans ce système de notation, la chaîne de chiffres $[d_{n-1}\, d_{n-2} \ldots d_0]$ $(d_i \in \{\bar{1}, 0, 1\})$ représentera le nombre D égal à $\sum_{i=0}^{n-1} d_i 2^i$.

Par exemple, on peut écrire le chiffre 5 de toutes les manières suivantes : [0101], [011$\bar{1}$], [1$\bar{1}$01], [1$\bar{1}\bar{1}$1] et [10$\bar{1}\bar{1}$], ce qui illustre le côté redondant d'un tel système de notation.

Tout nombre compris au sens large entre -2^n+1 et 2^n-1 peut s'écrire sur n chiffres (mais de manière non unique) dans ce système. La conversion entre l'écriture usuelle des nombres en base 2 et l'écriture à chiffres signés est triviale, en effet :

– Un nombre codé $[d_{n-1}\, d_{n-2} \ldots d_0]$ en écriture usuelle aura le même codage en « chiffres signés ».

– Pour coder en notation usuelle un nombre D qui s'écrit $[d_{n-1}\, d_{n-2} \ldots d_0]$ en « chiffres signés » il suffit d'effectuer (en notation usuelle) la soustraction $D^+ - D^-$, où le i-ième bit de D^+ est égal à d_i si d_i est positif et à zéro sinon, et où le i-ième bit de D^- est égal à $-d_i$ si d_i est négatif et à zéro sinon.

4.3. Algorithme d'addition associé

Supposons que nous désirions additionner deux nombres codés sur n chiffres dans la notation « chiffres signés » :

$$A=[A_{n-1}\, A_{n-2} \ldots A_0], \qquad B=[B_{n-1}\, B_{n-2} \ldots B_0]$$

Afin d'obtenir leur somme, codée dans le même système de notation :

$$Z=[Z_{n-1}\, Z_{n-2} \ldots Z_0]$$

On peut montrer (*voir* [CR78] ou [TYY85]) que l'algorithme suivant permet de calculer les chiffres S_i sans aucune propagation de retenue.

A_i	B_i	$A_{i-1}\, B_{i-1}$	C_i	S_i
$\bar{1}$	$\bar{1}$	Quelconques	$\bar{1}$	0
1	1	Quelconques	1	0
0 1 $\bar{1}$	0 $\bar{1}$ 1	Quelconques	0	0
1 0	0 1	De somme >0	1	$\bar{1}$
		De somme $\leqq 0$	0	1
$\bar{1}$ 0	0 $\bar{1}$	De somme <0	$\bar{1}$	1
		De somme $\geqq 0$	0	$\bar{1}$

Étape 1. – On calcule des termes C_i et S_i en se guidant sur le tableau précédent.

Étape 2. – En posant $S_n = C_{-1} = 0$, les chiffres du résultat Z de l'addition sont les Z_i, $i=0, 1, \ldots, n$, donnés par la relation :

$$Z_i = S_i + C_{i-1}$$

4.4. Un multiplieur série « poids forts d'abord »

Ce multiplieur est basé sur une idée originale due à Ercegovac et Trivedi [ET77], qui appellent *On-line* ce mode d'acheminement des données en série et en commençant par les poids forts [ERC84]. Remarquons au passage que dans [EL87] Ercegovac et Lang proposent un algorithme de conversion « au vol » et *On-line* de la notation « chiffres signés » vers la notation usuelle en complément à deux. Soient X et Y les opérandes que nous désirons multiplier, amenées « en série », chiffre par chiffre à partir des poids forts, et soit P leur produit, écrits en chiffres-signés sous la forme :

$$X=[x_{n-1}\, x_{n-2} \ldots x_0],$$
$$Y=[y_{n-1}\, y_{n-2} \ldots y_0],$$
$$P=[p_{2n-1}\, p_{2n-2} \ldots p_0].$$

Si l'on prend comme unité de temps le délai entre l'arrivée de deux chiffres consécutifs, au temps $n-j$ on dispose des chiffres $x_{n-1}\, x_{n-2} \ldots x_j$ et $y_{n-1}\, y_{n-2} \ldots y_j$.

Définissons les quantités $X^{(j)}$, $Y^{(j)}$ et $P^{(j)}$ comme suit (on désire évaluer $P^{(0)}$) :

$$\begin{cases} X^{(j)}=[x_{n-1}\, x_{n-2} \ldots x_j]=\sum_{i=j}^{n-1} x_i 2^{i-j} \\ Y^{(j)}=[y_{n-1}\, y_{n-2} \ldots y_j]=\sum_{i=j}^{n-1} y_i 2^{i-j} \\ P^{(j)}=X^{(j)}\, Y^{(j)} \end{cases}$$

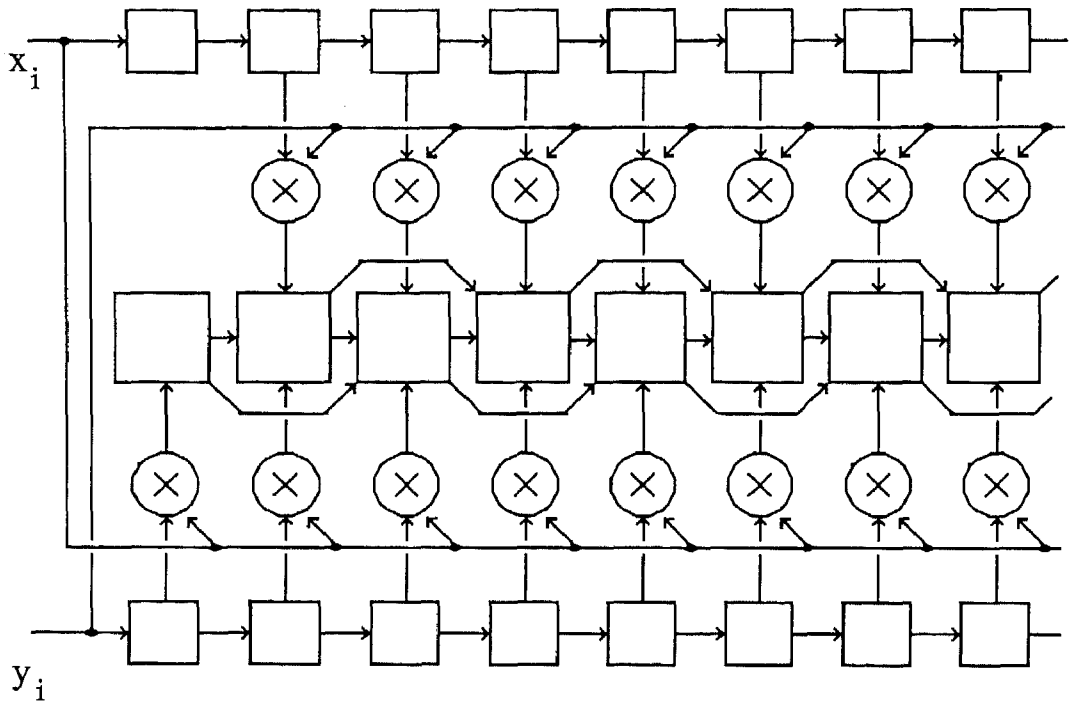
Il vient :

$$\begin{cases} X^{(j-1)}=2X^{(j)}+x_{j-1} \\ Y^{(j-1)}=2Y^{(j)}+y_{j-1} \end{cases}$$

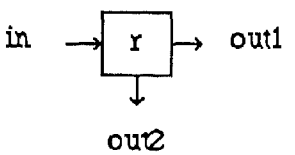
On peut donc, à partir de $P^{(n-1)}=X^{(n-1)}Y^{(n-1)}=x_{n-1}y_{n-1}$ calculer de proche en proche $P^{(n-2)}, P^{(n-3)}, \ldots, P^{(0)}$, grâce à la relation de récurrence :

$$(R) \qquad P^{(j-1)}=4P^{(j)}+x_{j-1}Y^{(j-1)}+2y_{j-1}X^{(j)}.$$

En base 2, une multiplication par 4 se réduit à un *décalage* de deux positions, et une multiplication par 2 à un décalage d'une position. De plus, puisque les chiffres x_{j-1} et y_{j-1} valent $\bar{1}$, 0 ou 1, les produits $x_{j-1}Y^{(j-1)}$ et $y_{j-1}X^{(j)}$ peuvent s'effectuer très facilement. On peut donc construire un circuit capable d'effectuer les calculs de la relation (R). Celui que nous proposons figure 9 est composé de trois cellules de base : des cellules de *mémorisation*, qui accumulent les valeurs courantes des suites $X^{(j)}$ et $Y^{(j)}$, des cellules *d'addition*, qui effectuent à chaque pas les deux additions de (R) en utilisant l'algorithme du

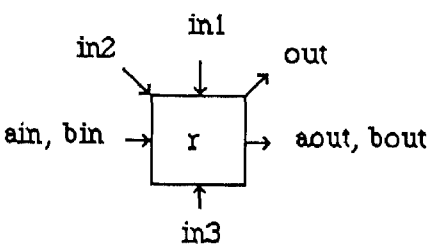


Cellule mémoire :

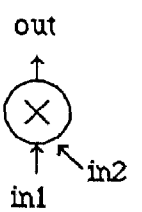


out1 := r
r := in
out2 := r

Cellule d'addition :



out := r
r := in2
aout := r
bout := in1
additionner r et in1 par la méthode présentée paragraphe 4.3, en tenant compte des chiffres précédents ain et bin. placer le résultat dans r
aout := r
bout := in3
additionner r et in3 par la méthode présentée paragraphe 4.3, en tenant compte des chiffres précédents ain et bin. placer le résultat dans r



Cellule de multiplication :

out := in1*in2
(N.B. in1 et in2 valent -1, 0 ou 1)

Fig. 9. — Le multiplieur série.

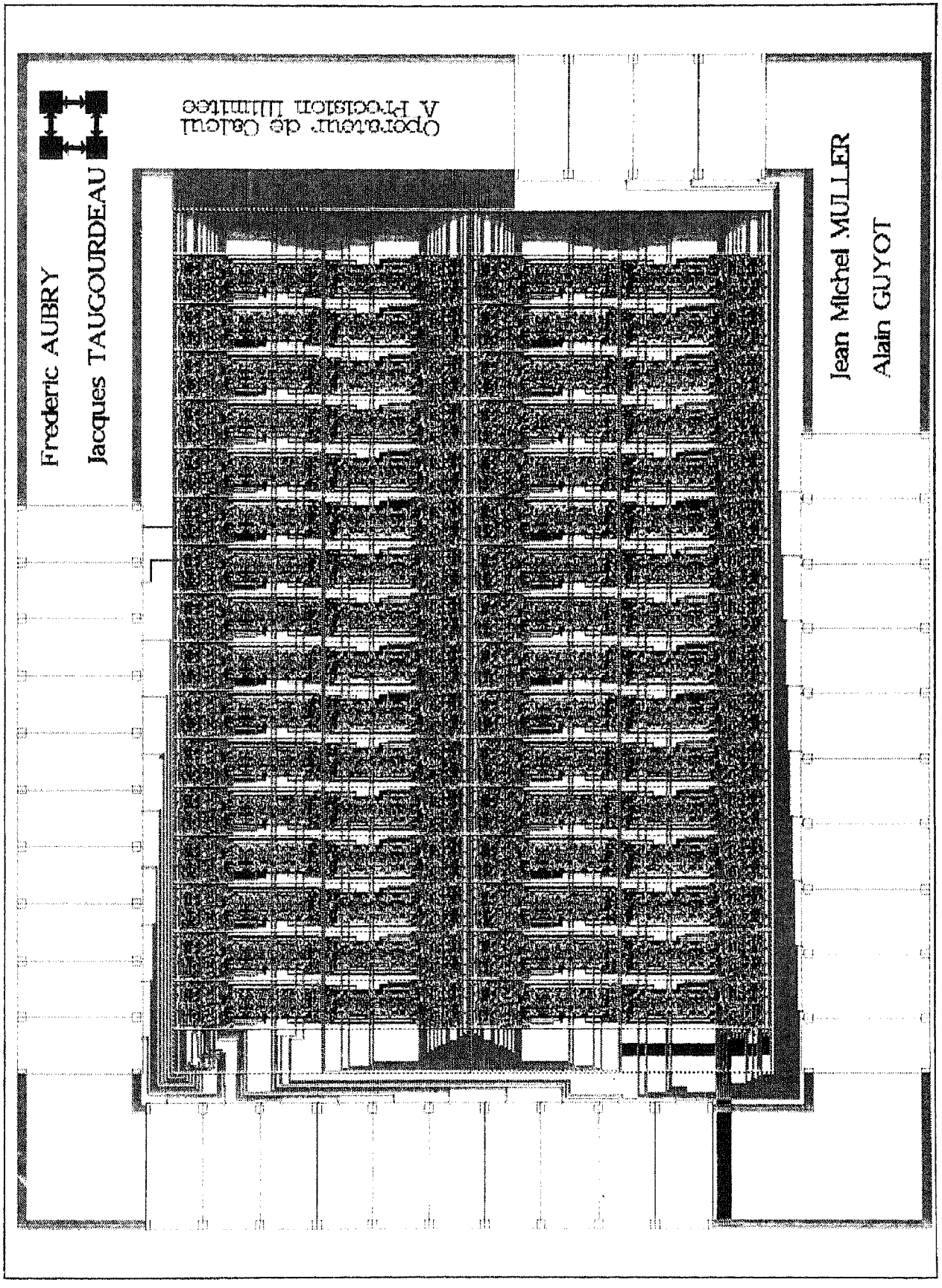


Fig. 10. — Dessin du circuit réalisant le multiplieur bit-série.

paragraphe 4.3, et des cellules de *multiplication*, capables de multiplier deux éléments de $\{\bar{1}, 0, 1\}$, utilisées pour évaluer les produits $x_{j-1}Y^{(j-1)}$ et $y_{j-1}X^{(j)}$. Adoptons comme unité de temps le délai nécessaire aux cellules d'addition pour effectuer leurs calculs. A chaque top de temps, le contenu de chaque cellule d'addition est transféré de *deux* positions vers la droite [ce qui correspond au calcul de $4P^{(j)}$ dans (R)], le contenu de chaque cellule mémoire est transféré d'une position vers la droite tandis qu'un nouveau chiffre est mémorisé dans les cellules de gauche (ce qui permet d'obtenir $X^{(i-1)}$ et $Y^{(i-1)}$ à partir de $X^{(i)}$ et $Y^{(i)}$). La figure 10 représente le dessin du circuit d'un tel multiplieur bit-série sur 64 chiffres, il a été réalisé au sein de l'équipe architecture du laboratoire TIM3. Le grand avantage de ce multiplieur est de fournir dès le premier top de temps le premier chiffre du résultat : si on effectue k produits successifs, les données du i-ième dépendant des résultats du $i-1$-ième (ce qui arrive dans le cas de la TFR, avec en plus des additions qui ne posent aucun problème en mode *On line*), ce multiplieur fournira le premier chiffre du résultat en un temps proportionnel à k, alors que le multiplieur de Lyon, par exemple, fournira le premier chiffre du résultat en un temps proportionnel à $n.k$ (n est la taille des opérandes).

4.5. Application a la conception d'une cellule série de TFR

Le multiplieur que nous venons d'étudier peut sans difficulté être utilisé pour construire une cellule élémentaire de TFR semblable à celle de la figure 2. On suppose que les puissances de ω sont calculées à l'avance et mémorisées. Tout comme ce multiplieur, une telle cellule travaillerait « en série » : elle recevrait ses données et fournirait des résultats chiffre par chiffre, en commençant par les poids forts.

Cette cellule doit être capable d'effectuer des sommes et des produits de nombres complexes.

Lorsque l'on désire effectuer le produit de deux nombres complexes $a+ib$ et $c+id$ (a, b, c et d réels), deux solutions sont possibles :

(1) Calculer des quatre produits partiels $p_1=ac$, $p_2=ad$, $p_3=bc$ et $p_4=bd$. Le résultat recherché vaut alors $(p_1-p_4)+i.(p_2+p_3)$.

(2) Calculer les trois produits partiels $p_1=ac$, $p_2=bd$ et $p_3=(a+b)(c+d)$. Le résultat recherché est $(p_1-p_2)+i.(p_3-p_2-p_1)$.

Dans la plupart des implantations, c'est la première solution qui est retenue car, même si elle nécessite une multiplication de plus que la deuxième, elle est

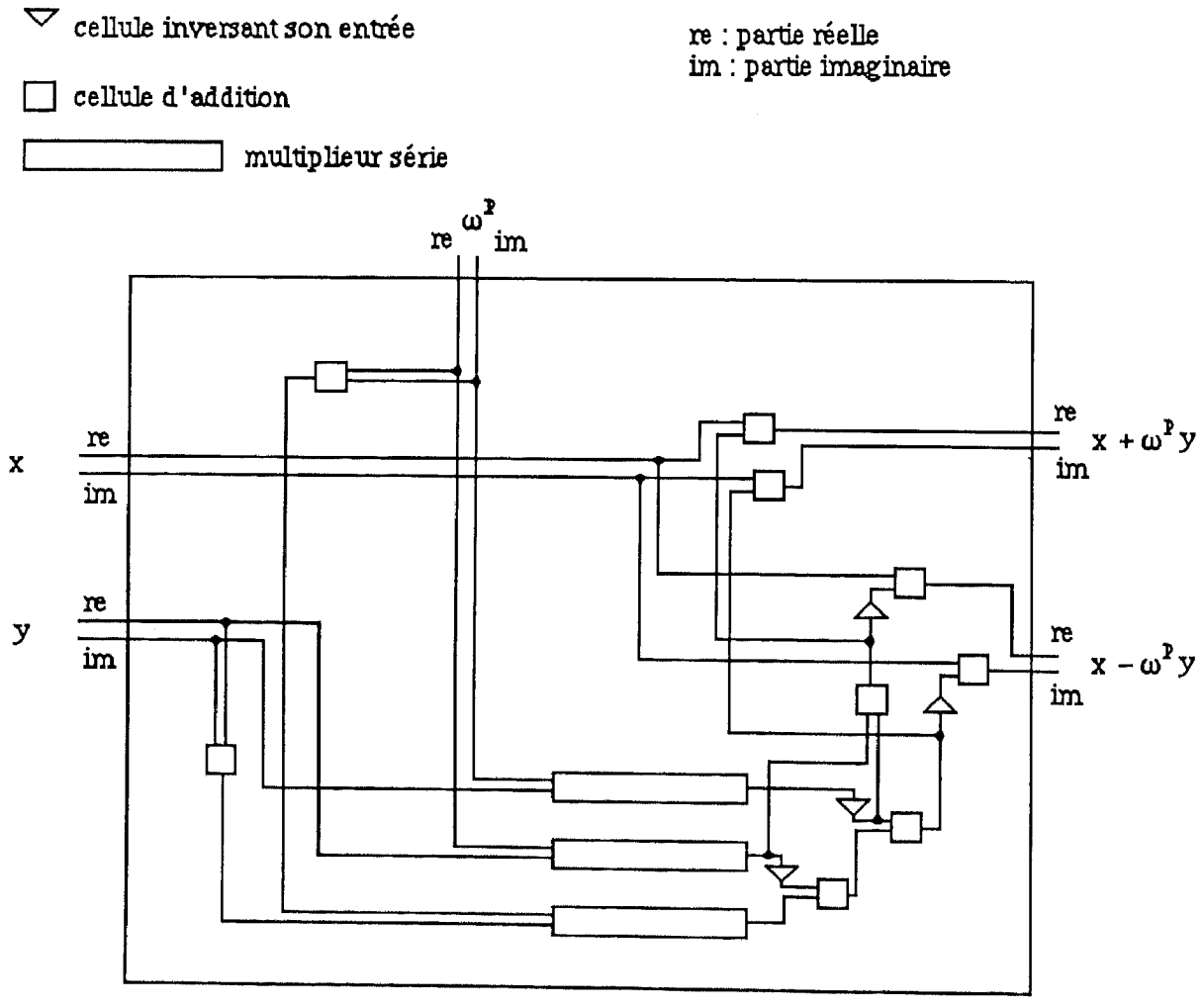


Fig. 11. — Cellule élémentaire de TFR.

tout de même moins coûteuse en additions. Dans le cas présent, les sommes s'effectuent particulièrement bien lorsque l'on travaille en série : en effet, une seule cellule élémentaire d'addition, utilisant l'algorithme décrit au paragraphe 4.3, est nécessaire. Nous aurons donc avantage à retenir la deuxième solution. La figure 11 décrit sommairement l'architecture d'une cellule élémentaire de TFR utilisant cette solution.

5. Circuit pour la TFR à taille variable

Le problème qui nous intéresse ici, est de calculer, de manière optimale, la TFR d'un grand nombre d'échantillons tout en gardant une architecture de taille réaliste. Nous reprenons l'idée de la TFR cyclique du chapitre 3 en fixant le nombre de cellules élémentaires pas trop élevé (8 par exemple) en utilisant le multiplieur bit-série précédent. Pour cela, nous utilisons une technique de partitionnement sur les entrées.

5.1. Partitionnement

Présentons cette technique sur un exemple simple. Supposons que l'on veuille calculer la TFR de 16 échantillons sur le réseau cyclique précédent de taille 8 (soit, 4 cellules). Puisque nous avons 16 échantillons, les puissances successives de ω qui interviennent sont 8, 4, 2 et 1, et elles apparaissent deux fois de suite puisque chaque étape dure 2 unités de temps (nous avons adopté pour les cellules la représentation de la figure 2).

Description formelle

Pour un problème général de taille $N = 2^p$ que nous cherchons à résoudre sur un réseau de 8 cellules (donc 16 entrées), on introduit conceptuellement à la sortie de chaque cellule deux rangées de registres de tailles 2^{p-4} dans lesquelles les éléments sont disposés au fur et à mesure des calculs (2^{p-4} vient de ce que les 2^p données initiales sont partitionnées sur 2^4 entrées). En réalité, ces registres sont composés d'un nombre fixe de cellules mémoire élémentaires qui agiront comme des tampons dans lesquels l'information circulera. Dès que la dernière série de calculs est terminée (ces calculs ont été menés avec $\omega^{2^{(p-1)}}$), on renvoie les éléments des registres (coefficients partiels) à travers le réseau cyclique (avec $\omega^{2^{(p-2)}}$) et ainsi de suite. Bien évidemment, les éléments d'une même rangée de registres auront à se rencontrer dans une phase finale (sur l'exemple précédent du calcul d'une TFR de taille 16 avec 4 cellules, la dernière colonne doit rencontrer l'avant dernière à la dernière étape). Ces échanges sont cablés et peuvent être anticipés lors du calcul des éléments des registres

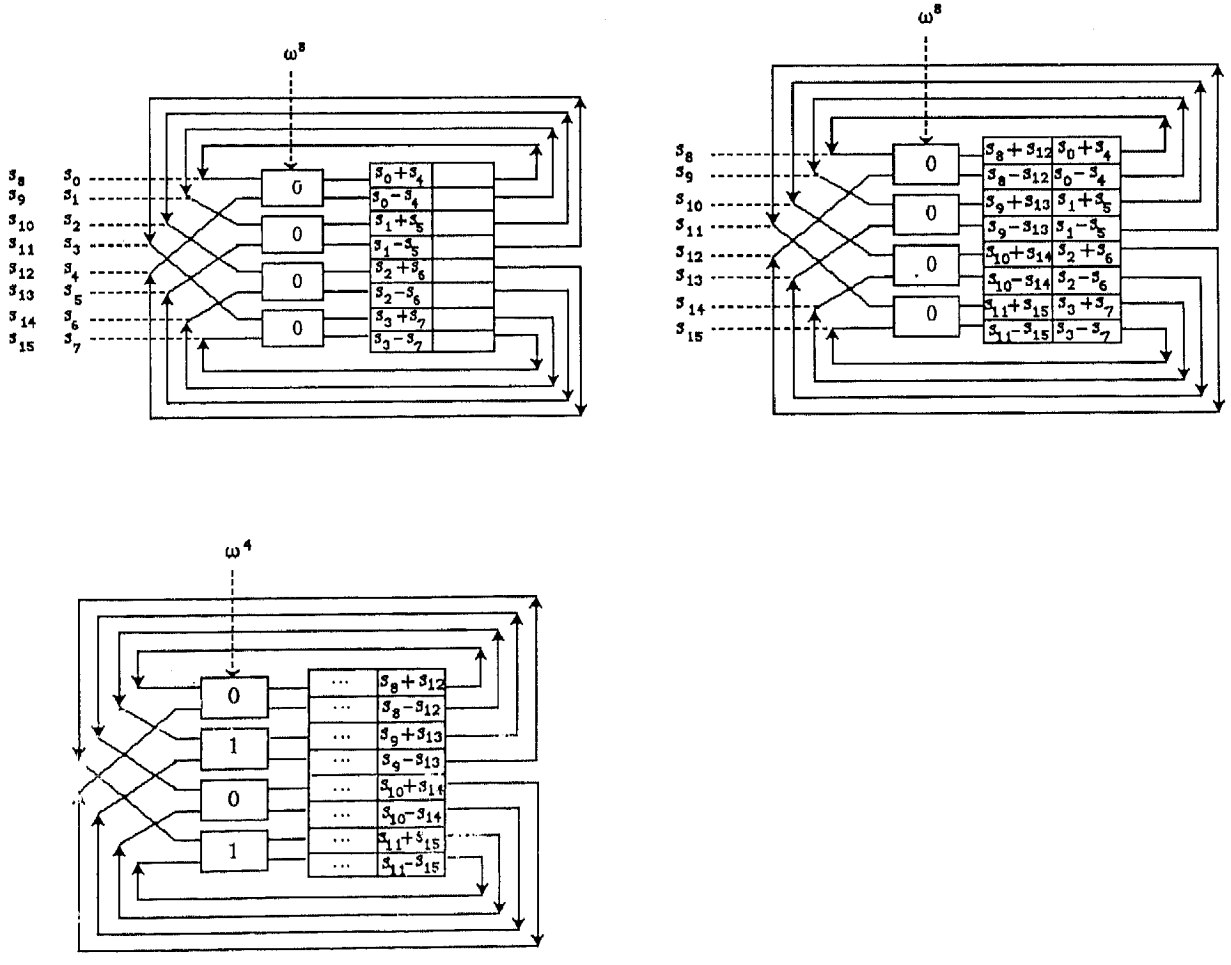


Fig. 12. — Principle du partitionnement.

non concernés de manière à ne jamais perdre de temps et ainsi rester optimal. Détaillons maintenant ce principe qui agit en fait comme un « perfect shuffle » sur les colonnes.

Au début, les colonnes sortant de la partie calcul dans l'ordre :

. . . -9-8-7-6-5-4-3-2-1 →

doivent y rentrer dans le même ordre. Juste après, le schéma de rentrée est le suivant :

. . . -9-8-6-7-5-4-2-3-1 →

et à l'étape suivante :

. . . -9-8-4-7-3-6-2-5-1 →

et ainsi de suite...

A ce niveau là, les coefficients partiels de Fourier sont répartis sur plusieurs étages des cellules mémoire.

5.2. Réalisation matérielle

Pour créer matériellement cette permutation de colonnes, on utilise un réseau de cellules de mémorisation, capables soit de lire une donnée in1 provenant de leurs voisines de gauche, soit de lire directement une donnée in2 sortant de la partie calcul. Pour chaque cellule, cela correspond pratiquement à adjoindre deux lignes de registres qui contiennent les coefficients intermédiaires apparaissant durant les calculs. Ces registres sont organisés en « FIFO » (premier entré-premier sorti). Dans une phase ultérieure, ces éléments auront à se rencontrer (en d'autres termes, une opération de type « perfect shuffle » doit être réalisée sur les colonnes). La difficulté est qu'ils sont à une distance variable : une commande globale C agissant sur toutes les cellules d'une même demi-colonne permet de ranger ces éléments en bonne place. Le registre interne est de la taille d'un coefficient complexe, et peut être alimenté chiffre par chiffre (c'est-à-dire sur 2 bits simultanément). Cette cellule est présentée figure 13.

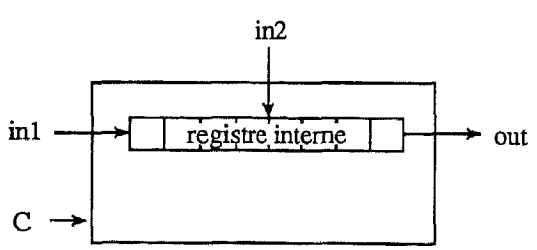


Fig. 13. — Cellule élémentaire.

si C=(0, 0) alors ne rien faire;

si C=(0, 1) alors on lit sur le bus (in1) et on décale;

si C=(1, 0) alors on lit depuis la gauche (in2) et on décale.

Les deux premières permutations de colonnes indiquées sur l'exemple ci-dessus se font alors à l'aide de la suite de commandes présentée figures 14 et 15. Il est important de noter que les suites de commandes associées à une colonne sont *périodiques*, il suffit donc de mémoriser le contenu d'une période (il n'est pas nécessaire de les calculer).

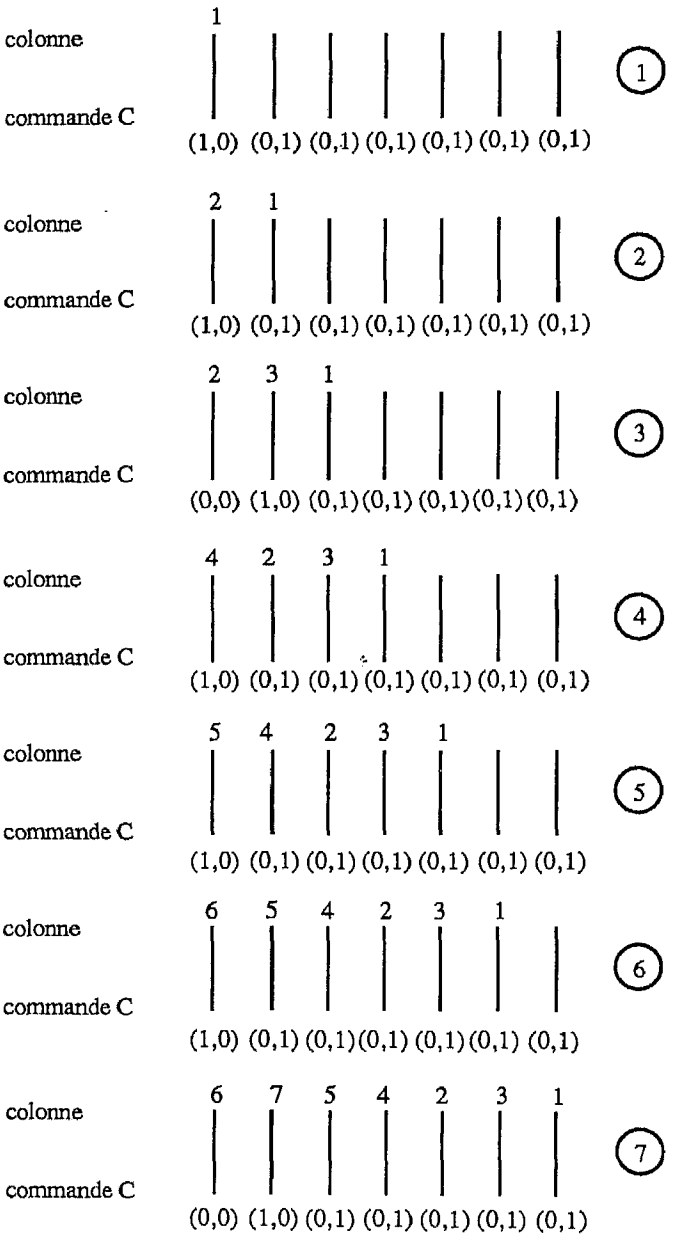


Fig. 14. — Première permutation : 9-8-6-7-5-4-2-3-1.

La figure 16 décrit tout le réseau de cellules de mémorisation. Grâce à la méthode de permutation de colonnes utilisée, les colonnes qui doivent se rencontrer dans le réseau TFR sont toujours les deux colonnes de droite. Des cellules tampons (les cellules B de la figure 16) permettent de renvoyer vers le réseau TFR, en tenant compte du « perfect-shuffle », les coefficients appropriés.

Ces cellules « B » permettent de conserver le flot continu de données propre à la circulation en « bit série », et évitent ainsi tout goulot d'étranglement.

La figure 18 représente le réseau complet composé de huit cellules de calcul (pour des raisons de clarté, les fils d'acheminement des commandes « C » ont été omis).

Résumons maintenant le fonctionnement complet du réseau pour 2^p d'échantillons. Nous supposerons

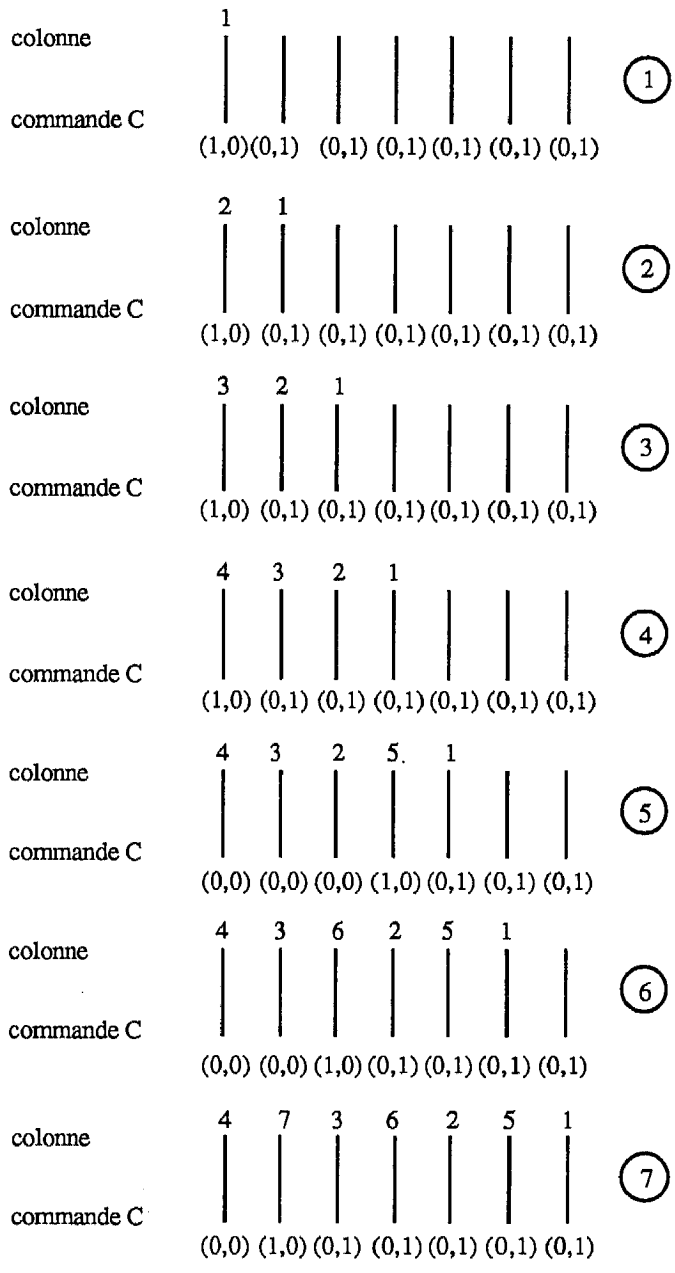


Fig. 15. — Deuxième permutation : 9-8-4-7-3-6-2-5-1.

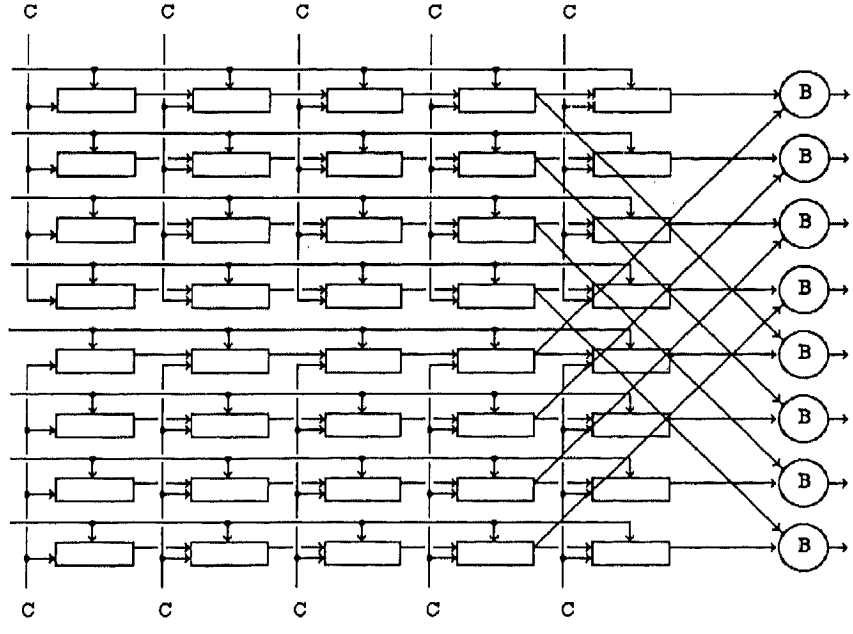
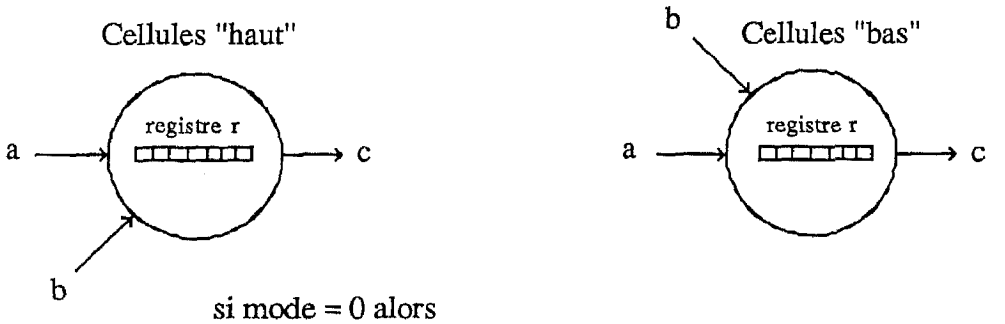


Fig. 16. — Description globale de la partie « commande ».




```
si mode = 0 alors
    début
        pour les cellules "haut" faire
            début
                c := a ;
                r := b ;
            fin ;
        pour les cellules "bas" faire
            début
              c := b ;
              r := a ;
            fin  ;
si mode = 1 alors pour toutes les cellules faire
  c := r.
```

Fig. 17. — Description et fonctionnement des cellules « B ».

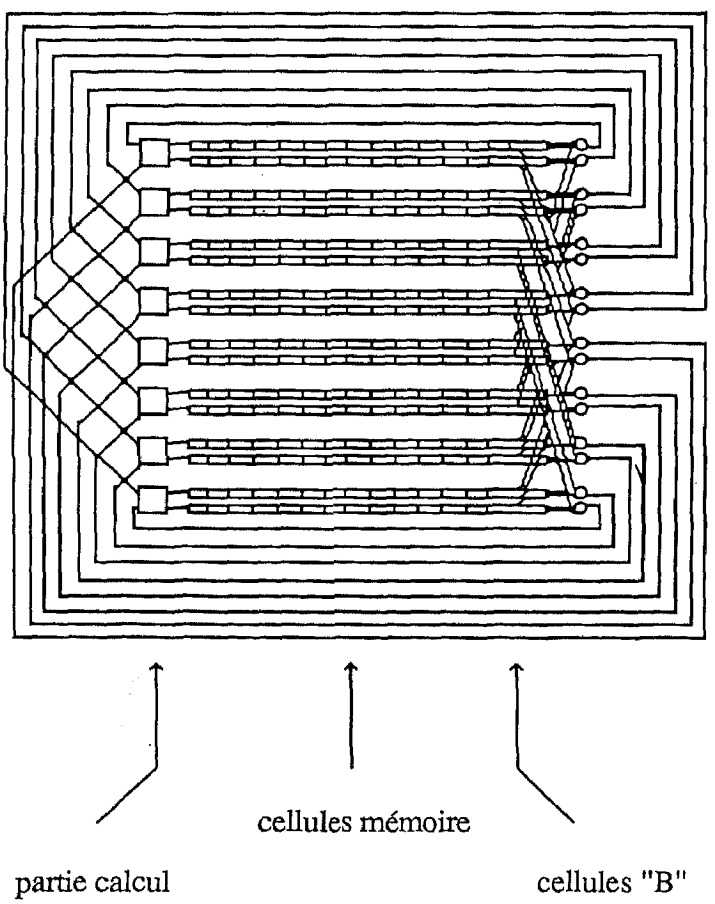


Fig. 18. — Le réseau complet.

que nous utilisons $m=2^q$ cellules élémentaires de TFR. L'algorithme procède en trois phases :

(1) *Initialisation des mémoires tampons :*
partitionner les 2^p entrées sur 2^{p-q-1} étages en entrées des m cellules
pour k : $=1$ jusqu'à 2^{p-q-1} faire
traiter directement les entrées à travers les cellules avec $\omega^{2(p-1)}$
ranger les résultats dans les 2 m lignes de cellules de mémorisation en 1° colonne et décaler les autres colonnes vers la droite.

(2) *calcul par le réseau cyclique :*
pour i : $=1$ jusqu'à q faire
pour k : $=1$ jusqu'à 2^{p-q-1} faire
traiter les intermédiaires avec le routage du réseau cyclique avec $\omega^{2(p-i-1)}$
ranger les résultats dans en 1° colonne des cellules de mémorisation

(3) *phase de rencontre des colonnes :*
pour i : $=1$ jusqu'à $p-q-1$ faire
pour k : $=1$ jusqu'à 2^{p-q-1} faire
traiter le couple de colonnes de droite sur le réseau cyclique avec $\omega^{2(p-q-1-i)}$
ranger les résultats suivant les commandes C décrites précédemment.

6. Conclusion

Nous avons proposé dans cet article un circuit spécialisé optimal permettant de résoudre le problème d'une Transformée de Fourier Discrète d'un grand nombre d'échantillons. Partant d'un réseau très efficace, cyclique, permettant le calcul de la TFR d'un nombre fixe d'échantillons, nous avons construit une cellule élémentaire qui traite les données en bit-série de manière continue, sans perte de temps. A partir d'un certain nombre de ces cellules, nous avons décrit une technique de partitionnement pour pouvoir résoudre des problèmes de taille quelconque. Si l'on prévoit de calculer la TFR de 2^p échantillons *au maximum* avec 8 cellules, il faudra 2^{p-4} colonnes de cellules de mémorisation.

Performances

Ce circuit est optimal dans le sens où il n'y a jamais de cellules de calcul inactives, réaliste quant à sa taille, combinant toutes les techniques modernes de l'architecture. Il est réaliste d'implanter 2^{10} colonnes de cellules de mémorisation, éventuellement sur un autre circuit, ce qui permet de calculer la transformée de Fourier discrète jusqu'à 16 384 échantillons.

Le calcul d'une transformée de Fourier discrète de 2^p échantillons sur k positions binaires demandera $k.p.2^p/8$ fois le temps de calcul d'une cellule élémentaire de multiplication binaire. Pour fixer un ordre de grandeur, si le temps de calcul d'une telle cellule est de l'ordre de 100 ns (ce qui correspond à une hypothèse réaliste), une TFR de 1 024 échantillons avec 32 bits de précision (ce qui correspond à 32 chiffres de notre écriture en « chiffres signés » des nombres, ce qui nécessite en fait 64 bits d'écriture) sera effectuée en environ 5 ms. Dans la pratique, pour des raisons de synchronisation, on sera obligé de placer des barrières temporelles tout au long du circuit, ce qui le ralentira, mais on peut donc espérer des temps de calculs inférieurs au centième de seconde.

Manuscrit reçu le 19 *janvier* 1988.

BIBLIOGRAPHIE

[AC86] R. C. Agarwal et J. W. Cooley, Fourier Transform and Convolution subroutines for the IBM 3090 VF, *IBM Journal of Research and development*, 30, n° 2, 1986.

[AHU74] A. V. Aho, J. E. Hopcroft et J. D. Ullman, *The design and analysis of computer algorithms*, Addison-Wesley publishing company, 1974, p. 251-275.

[AVI61] A. Avizienis, Signed-digit number representations for fast parallel arithmetic, *IRE Transactions on Electronic Computers*, 10, 1961, p. 389-400.

[BK81] R. P. Brent et H. T. Kung, The area-time complexity of binary multiplication, *J. of the ACM*, 28, n° 3, juillet 1981, p. 521-534.

[BON83] G. Bongiovanni, A VLSI Network for Variable Size FFT's, *IEEE Trans. on Computers*, C32, n° 8, August 1983, p. 756-760.

[CAV84] J. J. F. Cavanagh, *Digital computer arithmetic, design and implementation*, McGraw-Hill Computer Science Series, 1984.

[CW79] I. N. Chen et R. Willoner, An $O(n)$ parallel multiplier with bit-sequential input and output, *IEEE Trans. on Computers*, C-28, n° 10, octobre 1979.

[CR78] C. Y. Chow et J. E. Robertson, Logical design of a redundant binary adder, *Proc. 4th Symposium on Computer Arithmetic*, octobre 1978, p. 109-115.

[CT65] J. W. Cooley et J. W. Tuckey, An Algorithm for the Machine Calculation of Complex Fourier Series, *Math. of Comp.*, 19, 1965, p. 297-301.

[EL87] M. D. Ercegovac et T. Lang, On-the-fly conversion of redundant into conventional representations, *IEEE Trans. on Computers*, C-36, n° 7, juillet 1987.

[ERC84] M. D. Ercegovac, On-line arithmetic: an oveview, SPIE, *Real Time Signal Processing VII*, 45, 1984, p. 86-93.

[ET77] M. D. Ercegovac et K. S. Trivedi, On-line algorithmes for division and multiplication, *IEEE Trans. on Computers*, C-26, n° 7, juillet 1977, p. 681-687.

[GJ88] D. Gannon et W. Jalby, The influence of memory hierarchy on algorithm organization: programming FFT's on a vector multiprocessor, *Characteristics of parallel algorithms*, MIT press (à paraître).

[HWA79] K. Hwang, *Computer arithmetic principles, architecture and design*, New York, J. Wiley & Sons Inc., 1979.

[LYO76] R. F. Lyon, Two's complement pipeline multipliers, *IRE Trans. on Comm.*, COM-24, n° 4, avril 1976, p. 418-425.

[MC80] C. Mead et L. Conway, *Introduction to VLSI systems*, Addison-Wesley, 1980.

[PET83] W. P. PETERSEN, Vector Fortran for Numerical Problems on CRAY-1, *Com. of ACM*, 26, n° 11, 1983.

[REY83] J. F. REYSS-BRION, *Architecture de circuits spécialisés de traitement du signal*, RR352 Lab. IMAG, Grenoble, 1983.

[RS86] T. RHYNE et N. R. STRADER, A signed bit-sequential multiplier, *IEEE Trans. on Computers*, C-35, n° 10, octobre 1986.

[RK24] C. RUNGE et H. KONITZ, *Vorlesungen uber Numerisches Rechnen*, Springer Verlag, Berlin, 1924.

[SCO85] N. R. SCOTT, *Computer systems and arithmetic*, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1985.

[STO71] H. S. STONE, Parallel processing with the perfect shuffe, *IEEE Trans. on Computers*, C20, n° 2, février 1971.

[TAY87] F. J. TAYLOR, A Residue Arithmetic Implementation of the FFT, *Journal of Parallel and Distributed Comp.*, 4, 1987, p. 191-208.

[THO80] C. D. THOMPSON, A complexity theory for VLSI, *Ph. D. Dissertation*, Carneggie-Mellon University, Pittsburg, Pennsylvannie, août 1980.

[THO83] C. D. THOMSON, Fourier Transforms in VLSI, *IEEE Trans. on Computers*, C32, n° 11, November 1983, p. 1047-1057.

[TYY85] N. TAKAGI, H. YAKASURA et S. YAJIMA, High-Speed VLSI multiplication algorithm with a redundant binary addition tree, *IEEE Trans. on Computers*, C-34, n° 9, septembre 1985, p. 789-796.

[VUI83] J. E. VUILLEMIN, A Combinatorial limit to the computing power of VLSI circuits, *IEEE Trans. on Computers*, C-32, n° 3, mars 1983.

[WIN67] S. WINOGRAD, On the time required to perform multiplication, *J. of the ACM*, 14, n° 4, octobre 1967, p. 793-802.