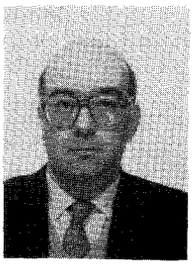
Vers un processeur de signal aisément utilisable

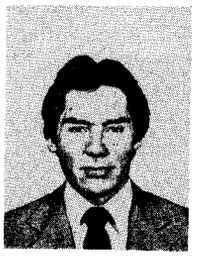
Towards an easy-to-use D.S.P.



J. OKSMAN

École Supérieure d'Électricité, Service des Mesures, Plateau de Moulon, 91192 GIF-SUR-YVETTE CEDEX.

Diplôme d'Ingénieur de l'École Supérieure d'Électricité, Professeur de Systèmes logiques, Électronique Numérique et Mini et Microinformatique, Responsable du groupe « Informatique pour l'Instrumentation » au Service des Mesures de l'ESE.



C. A. MARTINEZ GARCIA-MORENO

École Supérieure d'Électricité, Service des Mesures, Plateau de Moulon, 91192 GIF-SUR-YVETTE CEDEX.

Diplôme d'Ingénieur de l'École Supérieure de Génie Mécanique et Électrique de Mexico, Diplôme de spécialisation de l'École Supérieure d'Électricité, Thèse de Doctorat UPS-ESE (Conception d'une architecture VLSI de processeur de signal) au Service des Mesures de l'ESE. Actuellement à l'Institudo Politechnico National, Escuela Superior de Ingeniera Mecanica y Electrica, Mexico.

RÉSUMÉ

L'utilisation des processeurs de signaux n'est pas encore très courante, en dépit des avantages qu'ils procurent. Nous présentons ici un certain nombre de réflexions menées dans le sens d'une plus grande facilité d'utilisation de ces processeurs. Ces réflexions ont été concrétisées dans l'élaboration d'une architecture de processeur, à la fois optimisée et d'utilisation aisée.

MOTS CLÉS

Processeurs de signaux, architecture, applications.

SUMMARY

The use of digital signal processors is not yet commonly widespread despite their obvious advantages. In this paper we present a certain number of ideas helping ease their use. These ideas have been put to work by conceiving the architecture of a processor which is both optimal and easy to use.

KEY WORDS

Digital signal processors, architecture, applications.

1. Introduction

Les processeurs rapides dédiés au traitement numérique du signal (TNS) sont aujourd'hui largement utilisés dans un grand nombre d'applications. L'existence de processeurs sous forme monolithique VLSI aurait dû faire connaître au traitement du signal la même explosion que celle qu'a subie l'Informatique avec l'apparition des microprocesseurs monolithiques dans les années 70. En fait il n'en est rien, et les applications de ces processeurs se limitent à celles qui peuvent justifier d'un très grand nombre d'unités fabriquées (c'est le cas en particulier dans le domaine des télécommunications).

On peut expliquer cette situation par le fait qu'il y a une totale inadéquation entre les utilisateurs poten-

tiels (les « traiteurs de signaux »), qui ne sont pas en général (et ne doivent pas être...) des spécialistes de l'Informatique, et les outils qui leur sont offerts. En effet, les utilisateurs désirent bien évidemment que la machine qui va réaliser le traitement qu'ils ont défini soit très rapide, mais ils veulent aussi, et peut-être principalement, que celle-ci parle le « même langage » qu'eux. Ils veulent pouvoir implanter des algorithmes sans avoir à dominer de trop lourdes contingences informatiques [7].

A l'heure actuelle, que propose-t-on à ces utilisateurs? Dans les faits, ils disposent d'une part de logiciels de TNS plus ou moins faciles à utiliser, mais qui ne sont pas susceptibles de traiter en temps réel de vrais signaux, d'autre part de processeurs, effectivement très rapides [3], mais dans lesquels tout semble avoir été mis en œuvre pour décourager les utilisateurs non spécialistes. Parmi les principaux inconvénients de ces processeurs, on peut ainsi citer :

– leur microprogrammation, qui nécessite souvent des champs parallèles;

– l'absence de traitement direct des *objets* courants en TNS (des vecteurs en particulier);

– la faible puissance de leurs instructions, surtout pour des utilisateurs qui sont habitués, eux, à des primitives de haut niveau (opérations vectorielles, matricielles, etc.);

– la lourdeur du développement d'applications : appel à des outils logiciels complexes, utilisation de bibliothèques souvent mal adaptées, et surtout difficultés liées au composant lui-même : la mémoire qui va contenir le programme est souvent une mémoire de type ROM, qui ne peut être inscrite que chez le fondeur de silicium, ou alors n'existe sous forme accessible à l'utilisateur que dans des circuits « de développement » spéciaux, chers, et qu'il faut toujours adapter dans une application « normale ».

On conçoit donc que les seules applications où l'on trouve effectivement des processeurs rapides sont justement celles pour lesquelles le volume de la production a pu justifier les investissements nécessaires.

Le but de ce papier est de présenter quelques critères destinés à apporter des solutions à ces problèmes, ainsi qu'une architecture de processeur qui tente à la fois de maximiser les performances et de permettre une très grande facilité d'utilisation.

2. Le processeur « idéal »

2.1. Critère de performance

Il est évident qu'un processeur « idéal » doit être aussi rapide que le permet une technologie donnée. Il y a donc, à la base, un problème d'architecture. Comme on le constatera par la suite, ces problèmes ne sauraient se limiter à la seule partie opérative du processeur; on doit en effet également considérer l'architecture de l'unité de gestion du processeur (unité de « contrôle »), qui conditionne largement sa facilité d'utilisation. Le critère de rapidité demande lui-même à être précisé, aussi bien dans le domaine du TNS que dans celui des applications classiques. En effet, les processeurs existants sont souvent caractérisés par des quantités trompeuses, comme par exemple le « temps de cycle », qui, en TNS, correspond aussi bien à la durée d'une multiplication qu'à celle d'une addition. En fait, le « traiteur de signal » doit être vigilant, comme le montre l'exemple suivant, très fréquemment rencontré, en particulier dans les algorithmes de type « gradient » :

Soit à effectuer pour toutes les coordonnées E_i d'un vecteur E, l'opération suivante (appelée « correction », ou « adaptation », ou « combinaison linéaire », etc.) :

$$E_i = E_i + G \cdot L_i$$

(G étant un scalaire, L étant un vecteur).

Une telle opération doit être exécutée en un seul cycle du processeur, *y compris les actions annexes de passage à la coordonnée suivante et de test de fin de boucle* (puisqu'elles doivent être effectuées sur les N coordonnées du vecteur E). A titre d'exemple, un des processeurs les plus rapides du marché nécessite 5 cycles par coordonnée pour une telle opération. Il est alors évident que le seul critère lié à la technologie est insuffisant.

Mais la rapidité d'un traitement n'est pas seulement liée à la partie opérative du processeur. En effet, on sait que chaque fois qu'un processeur rapide (en fait tout circuit intégré) doit accéder à l'extérieur, la présence des capacités externes (qui sont beaucoup plus fortes que les capacités internes), ralentit considérablement l'électronique en augmentant les temps de propagation. Il est donc visible que l'utilisation même du processeur (et en particulier le fait que le programme en cours d'exécution soit à l'intérieur ou à l'extérieur) aura une influence sur les performances globales.

2.2. Critère de facilité d'utilisation

Un utilisateur potentiel est d'autant plus enclin à réaliser une application sur une machine de traitement de l'information que celle-ci possède les capacités :

– de traiter directement les objets qu'il manipule lui-même;

– de leur appliquer directement les opérations que demandent ces objets.

Prenons l'exemple trivial du « produit scalaire » de deux vecteurs (convolution, corrélation, etc.) : le « traiteur de signal » manipule fréquemment cette opération sur le papier. Lorsqu'il désire la réaliser avec un processeur rapide, il doit actuellement insérer l'action élémentaire (multiplication-accumulation) dans une boucle, gérer la fin de cette boucle, etc., puisque les processeurs classiques ne gèrent ni les objets (vecteurs) ni l'opération de composition envisagée (le produit scalaire).

Le cas particulier du TNS, et surtout des dispositifs qui traitent effectivement des signaux réels, se rapproche également du cas des microprocesseurs dans les équipements : il faudrait par exemple que le développeur puisse écrire ses programmes dans une mémoire classique, peu chère, avec un programmateur de PROM, ce qui exige que le composant-mémoire utilisé soit le plus simple possible (temps d'accès moyen, mot technologique de 8 bits, etc.). De plus, il faut

aussi résoudre le problème posé par la fabrication des programmes (on examinera par la suite une solution possible).

2.3. Critère d'intégrabilité

Ce critère est essentiel, car il agit directement sur les coûts du matériel développé. Il faut donc parfois refuser certaines qualités liées aux critères précédents afin que le processeur envisagé puisse être réalisé sous forme d'un circuit monolithique VLSI.

2.4. Conclusions

Au vu des critères précédents, un processeur à la fois réaliste et intéressant pourrait se définir par les éléments suivants :

– les objets de type « *vecteurs* » sont accessibles directement;
– des instructions de *haut niveau,* très puissantes et *optimisées* (au sens du nombre de cycles) sont directement exécutées par le matériel (techniquement, il s'agira de macro-instructions);
– ces macro-instructions sont dans une *mémoire externe,* qui peut donc être inscrite directement par l'utilisateur, et qui n'a pas besoin de performances extraordinaires (peu de contraintes sur le temps d'accès ou la largeur du mot-mémoire);
– elles réalisent des *primitives de traitement du signal* familières à l'utilisateur, comme par exemple l'instruction :

CONV H, E
(« convolution entre les vecteurs H et E »)

– indépendamment des macro-instructions qui sont implantées, on peut également faire effectuer n'importe quelle action au processeur, par exemple en plaçant des micro-instructions dans la mémoire externe.

On constate que, si les macro-instructions sont de haut niveau, leur nombre sera faible pour une application donnée : les programmes seront donc courts et simples et, comme le processeur aura peu fréquemment à aller chercher une de ces instructions dans la mémoire externe (cycles d' « extraction »), la perte de temps sera elle aussi négligeable (on montrera que l'on peut même l'annuler). *Rapidité* et *facilité d'utilisation* peuvent donc aller de pair.

La difficulté principale à surmonter consiste, comme on va le voir, à définir une architecture de gestion des macro-instructions qui ne grève pas les performances globales du processeur. En effet, le cas des processeurs de signaux est tout à fait différent de celui des machines informatiques microprogrammées. Dans notre cas, le passage des paramètres des macro-instructions est une exigence cruciale, comme une étude précédente l'avait montré [11].

En dernier lieu, il faudra évidemment que les performances strictement opératoires soient elles aussi optimales.

3. Optimisation des performances

3.1. Structure générale

L'architecture dite « de Harvard », qui a été historiquement la première utilisée dans les machines informatiques, est universellement admise pour les machines rapides car elle permet d'une part le recouvrement des extractions d'instructions et de leur exécution, d'autre part l'existence de mémoires séparées à commander en parallèle pour effectuer des accès multiples. En revanche, cette architecture et ces recouvrements entraînent les classiques difficultés de programmation, que l'on essayera donc de cacher à l'utilisateur « normal » [1].

La structure matérielle d'un tel processeur comprend alors classiquement une partie « opérative » (en anglais « data-path ») chargée des modifications de l'information et une partie de gestion (unité de « contrôle ») qui émet les ordres à destination de la précédente. Dans la partie opérative, nous distinguerons entre l'opérateur d'une part, et les mémoires et les bus de données de l'autre.

3.2. Opérateur

Pour permettre à coup sûr l'intégration du processeur, il faut aujourd'hui encore accepter de se limiter au traitement en virgule fixe de nombres réels (on double alors le nombre de cycles nécessaires à un calcul sur des variables complexes par rapport à un calcul similaire portant sur des variables réelles). La contrainte de représentation en virgule fixe n'est pas cruciale, dans la mesure on l'on dispose de suffisamment de dynamique de calcul avant débordement ou disparition des valeurs intermédiaires (« overflow » ou « underflow »). Pour cela, il faut que le nombre de bits du mot représentant une variable soit important, mais il n'a pas besoin d'être le même pour toutes les grandeurs intervenant dans un algorithme : il faut seulement qu'il soit maximal *là où c'est le plus utile,* c'est-à-dire lors des accumulations (somme de sous-produits) [12]. Pour cela, une largeur de 40 bits apparaît suffisante (elle correspond par exemple à 256 accumulations de produits de nombres de 16 bits). En revanche, les autres variables peuvent toujours être représentées sur 16 bits (en particulier, le mot « technologique », qui correspond à la largeur des bus d'échanges et à celle du mot-mémoire, peut être lui aussi de 16 bits).

L'opérateur optimal pour les opérations visées doit pouvoir effectuer simultanément une multiplication et une addition (à transformer éventuellement en accumulation), comme on peut le constater dans l'opération classique d'adaptation (ce point est bien connu dans la littérature [9, 2, 4]). On notera la différence entre cet opérateur et un multiplieur-accumulateur classique.

Les 40 bits permettant au minimum 256 accumulations de produits de mots de 16 bits, il y a nécessité de recadrer le résultat, c'est-à-dire de le multiplier par un coefficient constant, que l'on peut, sans trop nuire à la généralité, choisir égal à une puissance de 2. Ceci est généralement effectué par un décaleur programma-

ble placé en sortie de l'opérateur. De la même façon, il est bon dans certains cas de disposer d'un tel décaleur en entrée de l'opérateur, par exemple pour ajouter des composantes constantes homogènes soit à une entrée, soit à une sortie de l'opérateur [10].

Toute opération étant susceptible de débordement, il faut disposer d'un moyen de traiter ce cas. Deux scénarios sont possibles :

– soit l'on cherche à effectuer une opération en précision multiple : il faut alors pouvoir reprendre le résultat brut de l'opération;

– soit l'on désire que le signe du résultat obtenu soit conservé : il faut donc effectuer une *saturation* de ce résultat, en lui affectant la valeur la plus positive ou la plus négative possible. Cela se traduit par une non-linéarité qui, à condition d'être peu fréquente, est souvent tolérable dans beaucoup d'algorithmes [8].

3.3. Partie opérative

Le problème principal consiste à pouvoir alimenter l'opérateur et les mémoires avec suffisamment d'informations pour qu'ils puissent toujours être actifs (il s'agit là d'un point particulièrement important en architecture de machines). Dans le cas des algorithmes rapides, la solution suivante est déjà considérée comme optimale [3, 6] : elle consiste à implanter trois blocs mémoires indépendants. L'exemple de l'adaptation :

$$E_{i\,(\text{nouveau})} = E_{i\,(\text{ancien})} + G \cdot L_i$$

montre que l'on peut alors effectuer deux accès en lecture ($E_{i\,(\text{ancien})}$ et L_i) et un accès en écriture ($E_{i\,(\text{nouveau})}$) simultanément, si l'on accepte que ces accès soient entrelacés entre deux blocs-mémoires identiques (ping-pong), l'écriture se faisant sur un bloc, la lecture sur l'autre, les rôles des deux blocs étant échangés lors de l'opération d'adaptation suivante.

Dans les processeurs classiques, l'un de ces blocs est constitué de mémoire ROM et contient les coefficients constants d'une application (par exemple ceux de réponses impulsionnelles fixes), il est alors appelé « mémoire de coefficients ». En revanche, l'impératif de facilité d'utilisation impose que ce bloc-mémoire soit de type RAM dans un processeur conçu pour répondre à ce critère. Il faudra donc résoudre le problème posé par l'initialisation de cette mémoire aux bonnes valeurs des coefficients (*cf. infra*). Dans le cas du filtrage adaptatif, un autre intérêt de cette solution est qu'elle permet de ne pas faire de différence entre les filtres adaptatifs et les filtres fixes.

Nous avons vu plus haut que la partie opérative devait être capable de traiter directement les vecteurs rencontrés en TNS. Pour cela, il faut que les adresses des coordonnées puissent être calculées implicitement dans des calculateurs d'adresses sur chaque bloc-mémoire. L'expérience acquise dans ce domaine nous a montré qu'il était bon de considérer deux types de vecteurs : les vecteurs fixes et les vecteurs « glissants ». Un *vecteur fixe* est fixe en mémoire : son adresse de base est constante. En revanche, en TNS, on a souvent à traiter « le passé d'un signal » comme par exemple dans un filtre FIR :

$$S_n = \sum_{i=0}^{N-1} H_i \cdot E_{n-i}.$$

Dans ce cas, si on veut écrire $S_n = \langle H, E' \rangle$ (c'est-à-dire un produit scalaire entre deux vecteurs), il faut que le vecteur E′ ait ses coordonnées décalées dans le temps. Ceci peut être fait soit en les recopiant, ce qui demande N cycles (si N est la longueur), soit en incrémentant automatiquement l'adresse de base de E′, ce qui se fait en *un cycle*. Dans ce dernier cas, la mémoire étant de dimension bornée, le vecteur E′ (vecteur que l'on qualifiera de « *glissant* » [11]) doit être rendu circulaire, ce qui peut se faire par une division modulo K (K étant une constante strictement supérieure à N) de l'indice courant des coordonnées. On dispose alors d'un outil particulièrement puissant pour traiter tous les vecteurs qui apparaissent dans les algorithmes.

4. Programmation

4.1. Position du problème

La facilité d'utilisation d'un processeur rapide est liée en particulier à la possibilité d'écrire efficacement des programmes d'applications qui permettent d'exploiter au mieux les possibilités matérielles de la machine, principalement sous l'angle de la rapidité d'exécution. Mais écrire des programmes ne suffit pas : il faut aussi pouvoir les communiquer au processeur, et s'assurer qu'ils présentent le minimum d'erreurs. Une étude précédente [11] avait montré que beaucoup de solutions « classiques » ne permettaient pas de satisfaire aux impératifs ci-dessus. En particulier, les outils « logiciels » de création de programmes présentent tous une grave faiblesse quand il s'agit de faire exécuter le programme par le processeur. En effet, le résultat d'une telle génération de programme est une image-mémoire qu'il faut bien communiquer au processeur : on tombe alors sur des problèmes de nature de la mémoire-programme (RAM ou ROM) que l'on envisagera par la suite. En plus de cet inconvénient, les solutions de type « compilateur », par exemple, sont particulièrement peu efficaces car un compilateur, par essence, génère des sous-programmes : il se pose alors le problème (crucial) du passage des paramètres à ces sous-programmes. Un temps très important est ainsi perdu dans ces actions « inutiles » au sens du TNS. A titre d'exemple, un compilateur C livré avec un processeur disponible sur le marché semble pénaliser les performances de rapidité dans un rapport supérieur à 5 !

De la même façon, on pourrait montrer que la plupart des autres solutions (interpréteur, primitives enregistrées en ROM sous forme de procédures, macrogénérateurs spécialisés, etc.) ne répondent qu'imparfaitement à l'utilisation d'un processeur intégrable.

4.2. Architecture et programmation

L'architecture Harvard nécessite que des micro-instructions gèrent les différentes composantes de la par-

tie opérative. Normalement, ces micro-instructions sont en ROM interne, ce qui est une solution très rapide, mais excessivement lourde à mettre en œuvre pour le développement. Si l'on place ces microprogrammes en ROM (ou PROM) externe, le développement est plus aisé mais, comme les accès à l'extérieur d'un circuit intégré sont lents par rapport au fonctionnement interne, et que les micro-instructions sont de grande largeur (en général plusieurs dizaines de bits), cette solution est aussi à rejeter. Une solution possible pourrait consister à placer en ROM/PROM externe un programme qui soit recopié en RAM interne à la mise sous tension. Une telle solution nécessite une RAM interne de grande taille (alors que le point-mémoire RAM est beaucoup plus grand que le point ROM), entraîne une duplication du programme peu satisfaisante, et surtout impose comme les autres l'utilisation de micro-instructions.

On songe donc naturellement à des macro-instructions, situées à l'extérieur du processeur et appelant des suites de micro-instructions internes [5]. Cette solution tout à fait classique en informatique (machines « microprogrammées ») ne grève pas trop les performances, à la condition que le nombre de micro-instructions exécutées pour chaque macro-instruction soit grand (beaucoup de cycles internes par rapport aux cycles externes), ce qui sera le cas si ces macro-instructions sont *puissantes,* donc *utiles.* En revanche, il se pose un problème important : celui du passage des paramètres de ces instructions [11]. Nous allons le traiter sur un exemple : celui de la convolution de deux vecteurs.

Soit ainsi une macro-instruction du type :

$$S = CONV\ (V_1, V_2).$$

Les paramètres nécessaires sont :

- la longueur de V1 et V2;
- les adresses de V1 et V2 (voire les numéros des blocs-mémoires);
- l'adresse du résultat;
- les types (fixe, glissant, . . .) des vecteurs.

L'objectif, dans notre cas, est de ne pas avoir à transmettre systématiquement ces paramètres entre l'extérieur et l'intérieur du processeur : il faut donc que ces paramètres soient présents à l'intérieur avant l'exécution de la macro-instruction : pour ce faire il paraît pertinent de définir une phase de *configuration de tous les objets* qui interviennent dans une application. Cette phase peut avoir lieu une fois pour toutes, en général à la mise sous tension, en utilisant des *macro-instructions de définition des divers objets* (ces macro-instructions sont spécifiques de cette phase, et doivent donc être créées à cet effet, elles ont aussi comme rôle d'initialiser les coefficients constants du traitement). Lors de la boucle principale d'un algorithme « temps réel », une macro-instruction n'a plus à contenir qu'un pointeur très court désignant la zone contenant les paramètres de l'objet traité, inscrits dans la phase de définition. L'organigramme-type d'une application sera donc systématiquement celui représenté figure 1. On peut y vérifier que, dans la phase d'exécution, la période d'échantillonnage, en particulier, n'est plus influencée par les temps nécessaires au passage des différents paramètres.

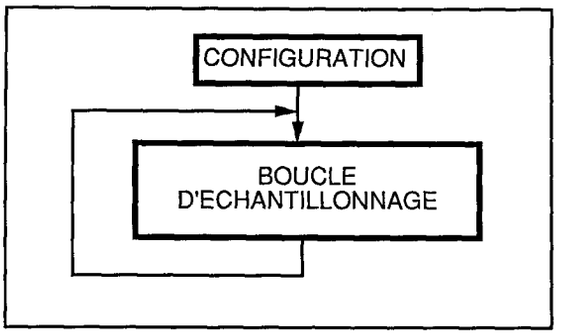


Fig. 1. – Organigramme d'une application.

Si les macro-instructions sont puissantes, elles ne seront que peu fréquemment extraites de la mémoire externe qui les contient. Afin d'éliminer le temps nécessaire à ces extractions (cycles de mémoire externe), on peut même envisager d'utiliser une unité d'interface avec cette mémoire qui permette le *recouvrement* entre ces cycles et l'exécution de la primitive courante. On pourra également profiter de ce dispositif pour utiliser une mémoire externe « classique » de 8 bits de large, les cycles nécessaires à l'extraction d'une macro-instruction (qui s'étend en général sur un certain nombre d'octets), étant alors invisibles dans la grande majorité des cas.

Par commodité, et pour éviter de figer le système, on doit aussi avoir la possibilité de placer des micro-instructions en mémoire externe, ceci dans le but d'augmenter la souplesse d'utilisation (on peut utiliser le processeur même si la primitive désirée n'est pas implantée), ou tout simplement pour essayer une nouvelle primitive qui sera par la suite intégrée elle aussi au processeur. Ce degré de liberté, particulièrement intéressant, ralentit évidemment le fonctionnement du processeur s'il est utilisé systématiquement.

On voit donc l'intérêt des solutions précédentes en ce qui concerne la programmation et l'utilisation du processeur :

- les instructions étant puissantes, les programmes sont donc courts, simples à écrire, peu susceptibles d'erreurs, et faciles à mettre au point;
- ils peuvent être placés facilement dans des PROM classiques de coût très bas;
- une modification de paramètres (exemple : longueur d'un vecteur) peut être effectuée directement dans une PROM externe.

4.3. Primitives (macro-instructions)

Si l'on désire optimiser un processeur pour une certaine classe de traitements, il est bon de considérer quelles sont les opérations pour lesquelles le produit « fréquence d'apparition par puissance de traitement » est le plus crucial. En informatique générale, il a été démontré que les instructions correspondantes étaient peu nombreuses, et peu puissantes : c'est le concept « RISC », abondamment illustré dans la littérature. Dans notre cas, comme on va le voir, la même démarche aboutit au contraire à des instructions certes peu nombreuses, mais très puissantes, et dont l'exécution doit être optimisée.

Les primitives de traitement du signal à réaliser en premier sont celles que l'on rencontre « le plus souvent » dans les algorithmes pour lesquels le processeur a été optimisé. Une étude un peu systématique [6] a porté sur toute une série d'algorithmes, en particulier les algorithmes de type « gradient » et tous leurs avatars, ainsi que les différents filtrages, que l'on rencontre en télécommunication, en traitement du signal pour les mesures (l'exemple-type étant la déconvolution en temps réel utilisée pour corriger la réponse dynamique de capteurs). Cette étude a montré que les opérations les plus fréquentes, qui sont aussi celles qui imposent le plus de cycles d'exécution, sont :

– la convolution de deux vecteurs (ex. : filtrage FIR);

– l'adaptation d'un vecteur par un autre (ex. : « gradient », algorithmes adaptatifs, . . .);

– la « mise à l'heure » d'un vecteur (opérateur de retard appliqué à chacune de ses coordonnées).

Un processeur efficace doit donc exécuter toutes ces opérations en un nombre minimal de cycles, c'est-à-dire en *un seul cycle par coordonnées* pour les deux premières, et en un cycle *unique pour la dernière*. A titre d'illustration, on pourra noter l'amélioration

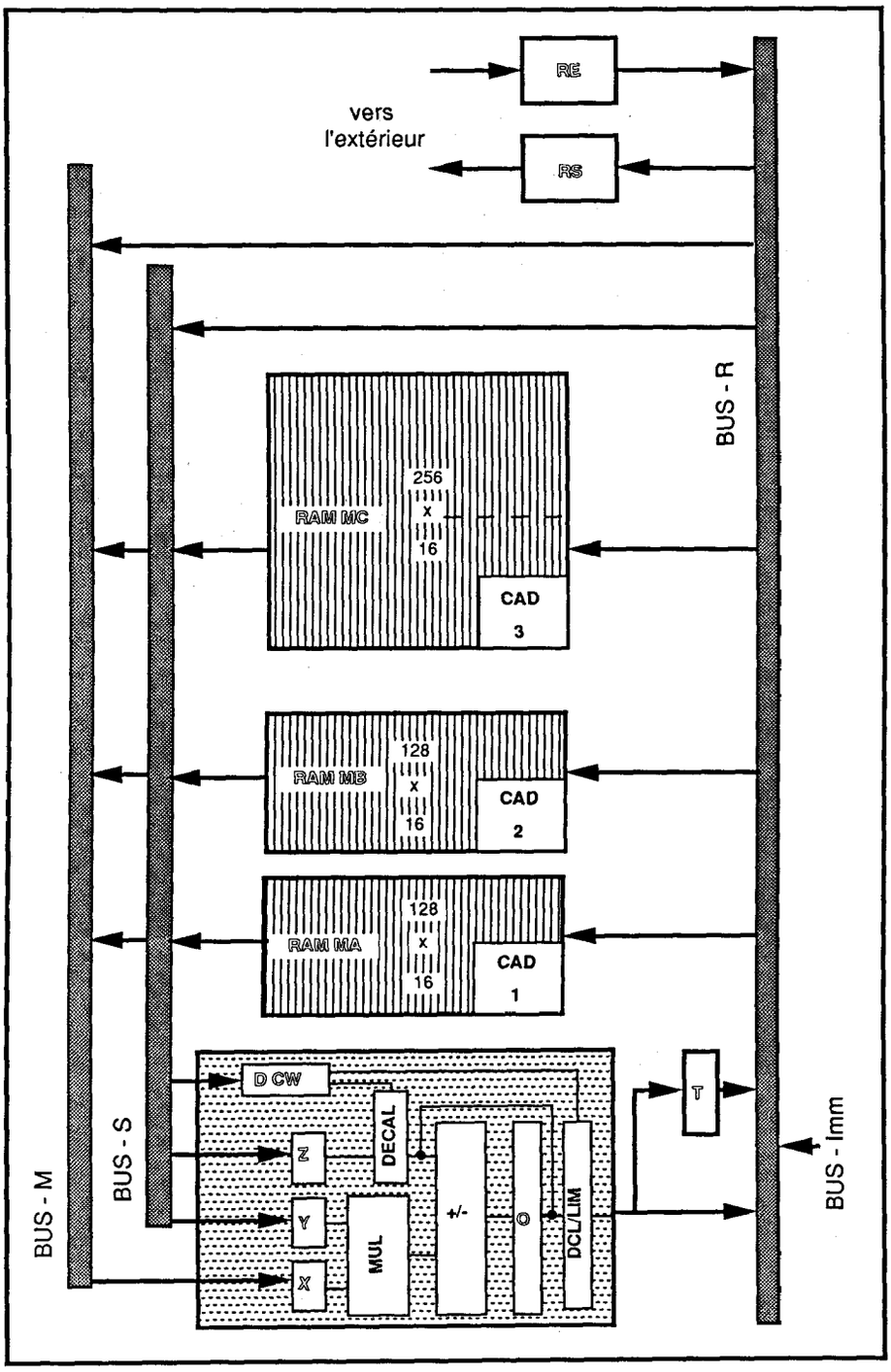


Fig. 2. — Partie opérative du processeur TRANSAM.

importante apportée par ce critère aux solutions proposées par la plupart des processeurs généraux de TNS.

Diverses autres primitives sont à envisager; on peut ainsi citer le module biquadratique, classique en filtrage récursif, et toutes les primitives permettant la configuration des objets à la mise sous tension (*cf. supra*).

D'autres primitives peuvent être éventuellement intégrées à un tel processeur, comme la FFT, qui n'est cependant pas une priorité dans les algorithmes performants d'aujourd'hui.

5. Le processeur « TRANSAM »

5.1. Architecture générale

Toutes les idées précédentes ont été utilisées dans la conception d'une architecture intégrable, baptisée processeur TRANSAM (pour TRAitement Numérique du Signal par une Architecture Macroprogrammable) [6]. La partie opérative et l'unité de contrôle (qui est celle qui permet la facilité d'utilisation) sont représentées sur les figures 2 et 3.

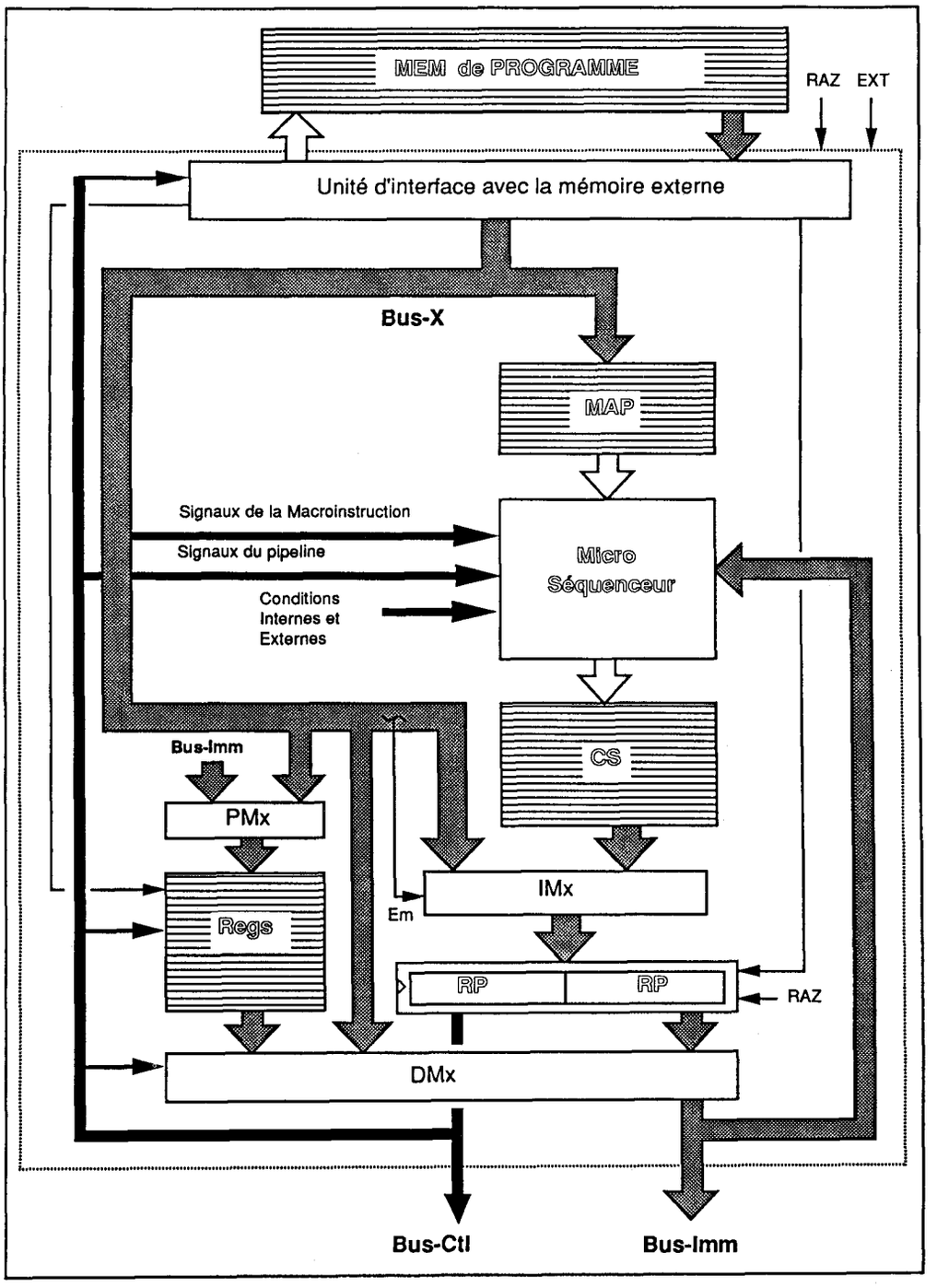


Fig. 3. — Unité de contrôle du processeur TRANSAM.

Cette partie opérative est classique : elle comprend trois mémoires de type « RAM », les deux premières (MA et MB) étant prévues pour travailler en « ping pong ». Les calculateurs d'adresse fournissent automatiquement l'adresse des données vectorielles, que les vecteurs soient fixes ou glissants (un opérateur « modulo 2^K » y est donc intégré). Ils permettent également la « mise à l'heure » d'un vecteur en un seul cycle.

L'opérateur correspond à la description de l'opérateur « optimal » vu plus haut (accumulations et sommes sur 40 bits, décaleurs en entrée et en sortie, limiteur, etc.).

L'unité de contrôle reprend les idées générales citées précédemment, on y notera :

– les registres destinés à contenir les paramètres des macro-instructions (REGS), ces registres sont chargés pendant la phase de configuration (*cf. supra*) avec toutes les valeurs nécessaires;

– l'unité d'interface permettant de rendre transparents les accès à la mémoire externe (ces accès se faisant par octets, la mémoire peut donc être à peu près quelconque; une macro-instruction, codée ici sur 32 bits, nécessite alors 4 extractions);

– le générateur d'adresse (MAP) de la mémoire de commande ROM (CS) qui contient les suites de micro-instructions;

– les multiplexeurs (IMX et DMX) permettant le mélange en mémoire externe de macro-instructions et de micro-instructions.

OPERATEURS GENERAUX DE TNS	
ADAPT_md1_DANS_md2	Adaptation d'un vecteur dans md1 et rangement du résultat dans md2.
CONV_md_C (code)	Convolution entre un vecteur dans md et le vecteur dans la mémoire C.
FILTRAGE NUMERIQUE RECURSIF	
INITIALISE_LE_FILTRE	Initialisation du filtre numérique.
BIQUAD (code)	Module biquadratique.
ENTREE_DU_FILTRE (valeur)	Entrée de l'échantillon dans le filtre.
LIAISON_ENTRE_BIQUADS (valeur)	Liaison entre modules biquadratiques
DECONVOLUTION DISCRETE EN TEMPS REEL	
INIT_LE_DECONVOL	Initialisation du déconvolueur.
ERRMOD_CAN (valeur)	Calcul de l'erreur du modèle.
PIVOT_md_CNA (code)	Sortie du pivot de md.
CONFIGURATION DU PROCESSEUR	
CONF_SDPn (code)	Configuration des pointeurs d'adresse secondaires.
CONFR_DPn	Configuration du pointeur de données "n".
CONFR_VECTEURS	Configuration des vecteurs de travail.
DEFR_n (valeur)	Définition du contenu du registre "n".
DONNEE_m (valeur)	Lecture d'une donnée dans le vecteur "m".
INITV_m (valeur)	Initialisation du vecteur "m".
DIVERS	
ATTENTE_PULSE_EXT	Attente d'une impulsion externe.
ATTENUATION (valeur)	Atténuation d'un résultat.
HALTE	Arrêt du macroprogramme.
MISE_A_LECHELLE (code)	Mise à l'échelle des résultats du calcul.
NOP	Non opération.
SAUT adresse	Branchement inconditionnel.
SORTIE (code)	Sortie d'un résultat.
END	Fin du programme.
EQUIV	Définition d'un symbole.
ETIQ	Définition d'une étiquette.

Fig. 4. — Primitives implantées.

5.2. Langage « utilisateur » (primitives implantées)

L'utilisateur a à sa disposition les primitives suivantes (*fig.* 4). Certaines sont tout à fait générales, d'autres ont été réalisées dans le cadre de certaines applications particulières.

L'adjonction d'autres primitives peut être réalisée sans problèmes (à titre d'exemple, seuls 254 mots, sur les 2 048 possibles, sont utilisés dans la mémoire de microprogramme du processeur).

L'exemple de programme suivant (*fig.* 5) montre bien la grande facilité de développement d'une application : il s'agit ici d'un simple filtre IIR (mise en cascade de modules biquadratiques).

```
0213 :    DEFR_10 (c_i_1_pour_4_BIQUADS)
0214 :    DEFR_11 (c_i_2_pour_4_BIQUADS)
0217 : ;
0218 :    INITIALISE_LE_FILTRE
0220 : ;
0223 :    DONNEE_C ( A01 )
0224 :    DONNEE_C ( A11 )
0225 :    DONNEE_C ( A21 )
0226 :    DONNEE_C ( MOINS_B11 )
0227 :    DONNEE_C ( MOINS_B21 )
0228 : ;
0229 :    DONNEE_C ( A02 )
0230 :    DONNEE_C ( A12 )
0231 :    DONNEE_C ( A22 )
0232 :    DONNEE_C ( MOINS_B12 )
0233 :    DONNEE_C ( MOINS_B22 )
0234 : ;
0235 :    DONNEE_C ( A03 )
0236 :    DONNEE_C ( A13 )
0237 :    DONNEE_C ( A23 )
0238 :    DONNEE_C ( MOINS_B13 )
0239 :    DONNEE_C ( MOINS_B23 )
0240 : ;
0241 :    DONNEE_C ( A04 )
0242 :    DONNEE_C ( A14 )
0243 :    DONNEE_C ( A24 )
0244 :    DONNEE_C ( MOINS_B14 )
0245 :    DONNEE_C ( MOINS_B24 )
0247 : ;
0249 : ETIQ BOUCLE_ECHANTILLONNAGE
0251 : ;
0252 :        ATTENTE_PULSE_EXT
0253 : ;
0258 :        ENTREE_DU_FILTRE (k_entree)        ; k =< 1.0
0260 :        BIQUAD (lim_x2)
0265 : ;
0266 :        LIAISON_ENTRE_BIQUADS (k_1_2)      ; k =< 1.0
0268 :        BIQUAD (lim_x2)
0273 : ;
0274 :        LIAISON_ENTRE_BIQUADS (k_2_3)      ; k =< 1.0
0276 :        BIQUAD (lim_x2)
0281 : ;
0282 :        LIAISON_ENTRE_BIQUADS (k_3_4)      ; k =< 1.0
0284 :        BIQUAD (lim_x2)
0285 : ;
0290 :        ATTENUATION (k_sortie)             ; k =< 1.0
0292 :        SORTIE (lim_x2)
0293 : ;
0295 :    SAUT BOUCLE_ECHANTILLONNAGE
0296 : ;
0297 :        END
```

Fig. 5. — Exemple de programme (filtre récursif).

5.3. Micro-instructions

Les micro-instructions servent à la gestion des parties opératives et de contrôle vues plus haut. Elles permettent bien évidemment d'exécuter en un *seul cycle* par coordonnées les opérations optimisées. Pour ce faire ont été en particulier implantés :

– les opérations classiques de branchement, y compris les tests sur conditions, internes ou externes (synchronisation);

– la gestion de l'opérateur et des mémoires permettant d'exécuter simultanément une opération (multiplication et/ou addition, multiplication-accumulation), une écriture, une lecture ainsi que les calculs d'adresses;
– un compteur de boucles intégré au séquenceur;
– les transferts à travers les bus;
– le chargement des registres de configuration.

5.4. Logiciels d'aide au développement

Il a été développé la totalité des outils utiles à la mise en œuvre de TRANSAM, c'est-à-dire :
– le traducteur en binaire des macro-instructions (qui permet donc d'obtenir une PROM-programme, par exemple);
– un simulateur du processeur, comprenant lui-même deux parties : d'une part la conversion entre micro-instructions et mémoire de commande, d'autre part une simulation du matériel. Ce simulateur a permis en particulier de valider l'architecture proposée sur une série d'applications, comme celle présentée plus loin. Cet outil peut être utilisé par un spécialiste pour concevoir soit de nouvelles primitives, soit un programme résidant en totalité dans la mémoire de commande (par exemple pour des applications de très grande série et pour lesquelles on souhaite économiser la mémoire externe);
– les outils de génération de signaux simulés et de mesures des résultats du traitement.

A titre d'exemple, une application de déconvolution en temps réel a été ainsi développée. Les figures 6 et 7 représentent la liste de la partie « active » de l'algorithme et un résultat obtenu grâce au simulateur.

```
0283 : ;*****************************************************************
0284 : ;
0285 : ;
0286 : ; COMMUTATION DES VECTEURS DE TRAVAIL.
0287 : ; CALCUL DE LA SORTIE ESTIMEE A PARTIR DU VECTEUR DE TRAVAIL (2)
0288 : ; ET DES "N" PREMIERES COORDONNEES DU VECTEUR DES CONSTANTS (3) :
0289 : ;
0290 :          CONV_B_C ( LIM_X1 )   ; CONVOLUTION = O = E2*H.
0291 : ;
0292 : ;
0293 : ; ATTENTE DU PULSE DE SYNCHRONISATION EXTERNE SUR LE BORNE EXT.
0294 : ; LECTURE DE LA VALEUR DE LA SORTIE REELLE SE TROUVANT AU CAN ET
0295 : ; CALCUL DE L'ERREUR DU MODELE :
0296 : ;
0297 :          ATTENTE_PULSE_EXT     ; FREQUENCE D'ECHANTILLONNAGE.
0298 : ;
0299 :          ERRMOD_CAN ( X_1/2 )  ; ERREUR = T = O - CAN
0300 : ;
0301 : ;
0302 : ; CORRECTION DES ENTREES ESTIMEES REPRESENTEES PAR LE VECTEUR DE
0303 : ; TRAVAIL (2) ET REMISE A L'HEURE DANS LE VECTEUR DE TRAVAIL (1):
0304 : ;
0305 :          ADAPT_B_DANS_A        ; E1 = E2 + L*ERR = E2 + L*T.
0306 : ;
0307 : ;
0308 : ; EXTRACTION DE L'ESTIMEE "PIVOT" CONTENUE DANS LE VECTEUR DE
0309 : ; TRAVAIL(1) :
0310 : ;
0311 :          PIVOT_A_CNA ( LIM_X2 ) ; SORTIE SUR RS.
0312 : ;
0313 : ;
0314 : ; FIN DE L'ITERATION. SAUT AU DEBUT DE LA BOUCLE :
0315 : ;
0316 :          SAUT  DEBUT_BOUCLE
0317 : ;
```

Fig. 6. — Exemple de programme : déconvolution en temps réel.

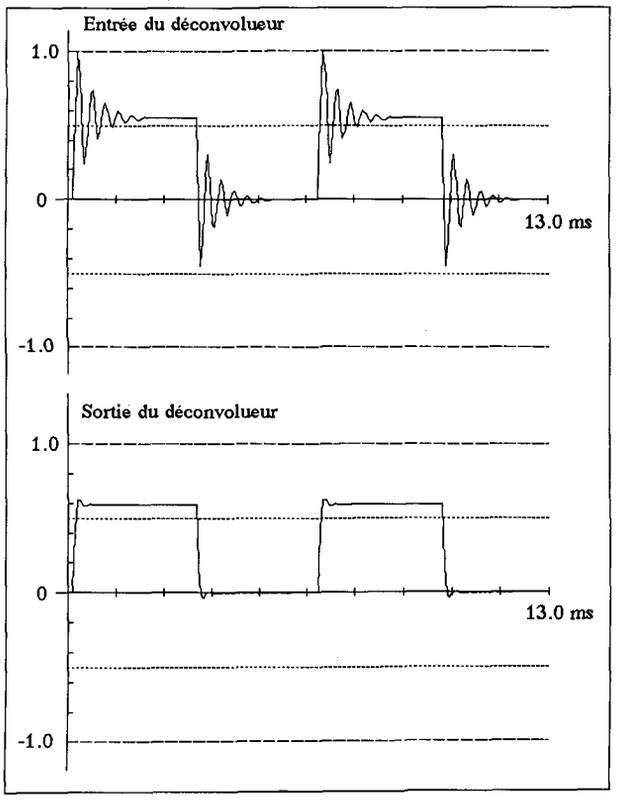


Fig. 7. — Exemple de résultats obtenus grâce au simulateur.

Sur la figure 6, on voit l'intérêt des solutions adoptées (*NB* les signes « ; » sont des marques de commentaires) : l'algorithme (décrit en commentaire) est dans ce cas traduit sous la forme d'*une seule macro-instruction par étape*. La phase de configuration (non représentée) a permis de rentrer la réponse impulsionnelle utilisée (sur 32 prises), les 32 coefficients du vecteur de correction et la valeur du pivot de l'algorithme (toutes les variables, sauf le résultat des convolutions, sont codées sur 16 bits). Avec un temps de cycle de 100 ns, la fréquence d'échantillonnage maximale serait de l'ordre de 60 kHz, ce qui permettrait de déconvoluer des signaux issus de systèmes (capteurs ou autres) dont la bande passante s'étend jusqu'à 15 kHz.

5.5. Exemple de VLSI

Le schéma externe de TRANSAM, obtenu en considérant le nombre de communications nécessaires avec l'extérieur, conduit à un circuit de 44 pattes classique. La densité d'intégration serait de l'ordre de 400 000 transistors, avec les données suivantes :
– mémoire de commande interne : 2 048 mots de 32 bits;
– mémoires de données : 2 × 128 mots de 16 bits;
– mémoire de coefficients : 256 mots de 16 bits.

En général, ce composant devrait être accompagné d'une mémoire quelconque contenant le programme (qui le plus souvent ne comporte que quelques dizaines d'octets, vu la puissance des primitives).

Avec une fréquence d'horloge de 10 MHz, tout à fait compatible avec les technologies CMOS actuelles les

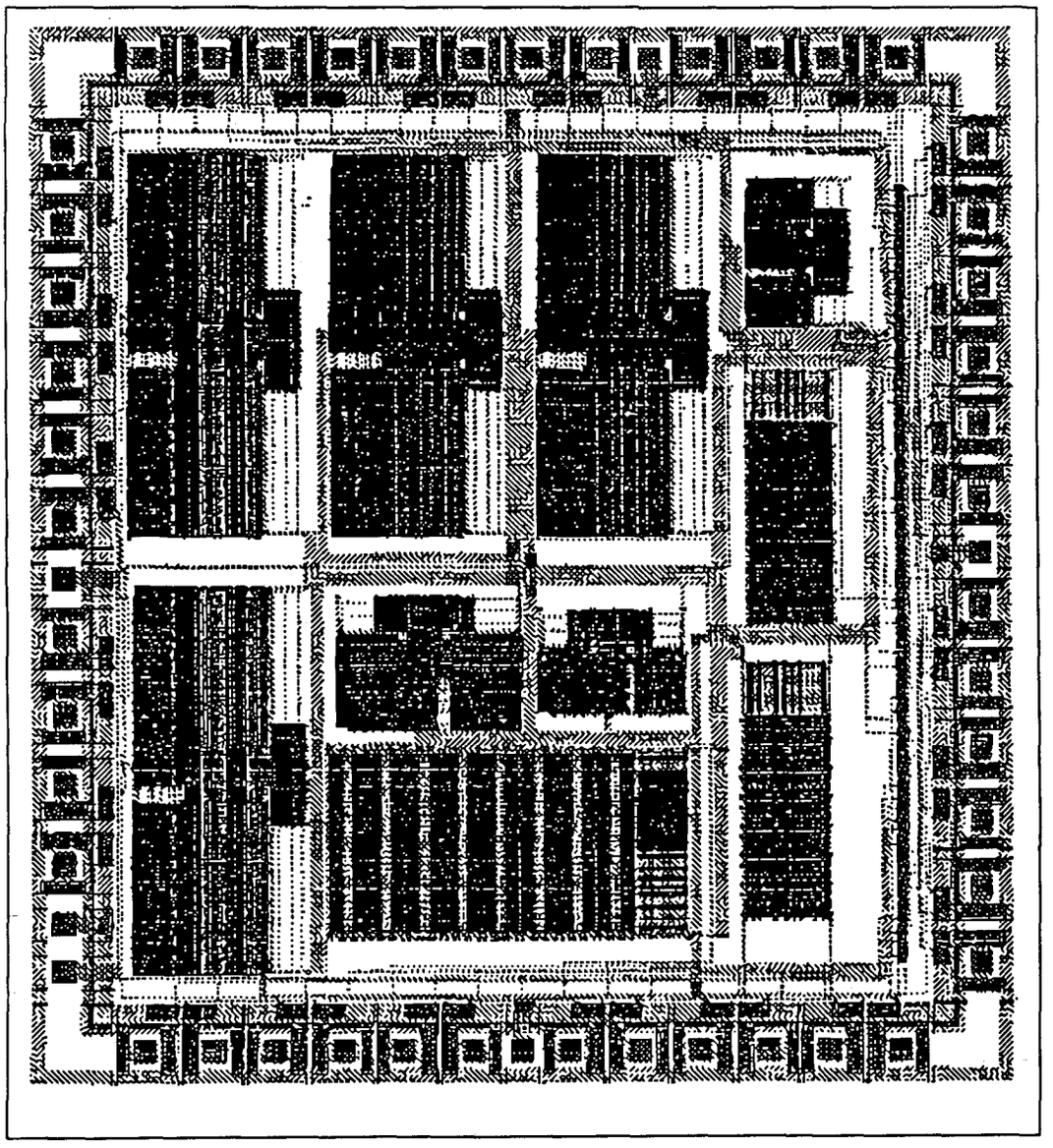


Fig. 8. — Évaluation de l'implantation de l'unité de contrôle.

moins dispendieuses, on aurait alors les ordres de grandeurs suivants pour les périodes d'échantillonnage de quelques traitements :

- filtrage récursif : 0,6 µs/biquad;
- déconvolution en temps réel (réponse sur 32 points) : 10 µs;
- égalisation réelle adaptative (64 prises) : 15 µs;
- annulation d'écho réelle (256 prises) : 60 µs.

Grâce à l'amabilité de la société VLSI Technologies, voici ce que pourrait être une implantation de la seule unité de contrôle (*fig.* 8).

6. Conclusion

Le processeur précédent n'a pas la prétention de résoudre tous les problèmes posés par la réalisation d'applications de TNS. En particulier, la partie opérative a été choisie de façon relativement simple afin d'assurer l'intégrabilité du produit. Elle pourra être améliorée, par exemple par l'optimisation du traitement des nombres complexes ou de la FFT, voire par le traitement de nombres en virgule flottante. De même, la conception du VLSI est encore une tâche de longue haleine. En revanche, il est plus que vraisemblable que l'architecture de programmation précédente permettrait de « démocratiser » le traitement du signal, autant que les simples microprocesseurs ont démocratisé les solutions informatiques dans la plupart des équipements réalisés jusqu'alors avec de l'électronique classique.

Manuscrit reçu le 12 *janvier* 1989.

BIBLIOGRAPHIE

[1] J. ALLEN, Computer architecture for signal processing, *Proceedings of the IEEE,* 63, n° 4, April 1975, p. 624-633.

[2] L. BUREAU, M. JABES, D. N'GUYEN, M. SARLOTTE et A. BEN-CHEIKH, *Conception de la partie opérative d'un processeur de traitement de signal.* Rapport interne, Service des Mesures de l'École Supérieure d'Électricité, 1987.

[3] Documentations techniques concernant les processeurs numériques de signal suivants :
AMI : 28211;
ANALOG DEVICES : ADSP-2100;
ATT : WE DSP16;
ATT : WE DSP32;
FUJITSU : MB 8764;
HITACHI : HD 61810;
INTEL : 2920;
NEC : uPD 7720;
NEC : uPD 77220;
TEXAS INSTRUMENTS : TMS 32010;
TEXAS INSTRUMENTS : TMS 320C25;
TEXAS INSTRUMENTS : TMS 320C30;
THOMSON : TS 68931-PSI.

[4] L. HETTE, M. LAZARUS, L. MAGIORI et F. MANCHON, *Conception de la partie opérative d'un processeur de traitement de signal,* Rapport interne, Service des Mesures de l'École Supérieure d'Électricité, 1985.

[5] S. S. HUSSON, *Microprogramming principles and practices,* Prentice Hall, New York, 1970.

[6] C. MARTINEZ GARCIA-MORENO, Conception d'une architecture de processeur de signal VLSI, programmable en langage évolué et optimale dans le traitement d'algorithmes rapides, *Thèse de doctorat,* Université de Paris-Sud/École Supérieure d'Électricité, 1988.

[7] J. MCLEOD, Digital Signal Processing: chips are here, but software isn't, *Electronics,* 31 March, 1988, p. 57-59.

[8] A. PELED et B. LIU, *Digital signal processing,* John Wiley & Sons, § 4.4, 1976.

[9] R. REYNAUD, F. DEVOS et E. MARTIN, Apport de l'intégration pour les processeurs de traitement de signal, Institut d'Électronique, Fac. des Sciences, Orsay, *Mesures,* numéro spécial, 4 novembre 1985, p. 21-23.

[10] R. SEARA, Conception et réalisation d'un système de traitement numérique du signal ayant comme principale application la déconvolution rapide en temps réel, *Thèse de Docteur-Ingénieur,* Université de Paris-Sud, Centre Orsay, 1984.

[11] SETRAAPS, *Exploitation d'un processeur rapide de traitement du signal. Système Automatique de Traitements Rapides d'algorithmes en Programmation Évoluée,* Rapport de fin de contrat, Service des Mesures de l'École Supérieure d'Électricité, 1982.

[12] J. THOMPSON et S. K. TEWKSBURY, LSI signal processor architecture for telecommunications applications, *IEEE Trans. Acoust., Speech, Signal Processing,* ASSP-30, n° 4, Aug. 1982, p. 613-631.