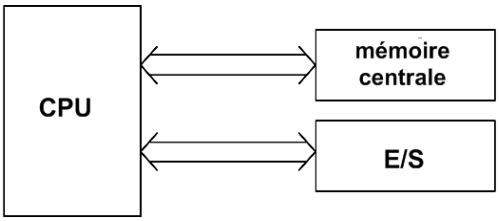
Sommaire

Chapitre 0 – Aperçu sur le microprocesseur

Un micro-processeur est une machine à états séquentielle qui peut effectuer la fonction de milliers de circuits logiques mais occupe beaucoup moins d'espace. Il est conçu pour être utilisé comme CPU d'un micro-ordinateur.

Un ordinateur peut être représenté sous une forme simplifiée par le schéma suivant



Unité centrale (CPU)

Elle contrôle le déroulement de l'ensemble des opérations qui s'effectuent dans le système complet.

Périphériques d'entrée/sortie (E / S)

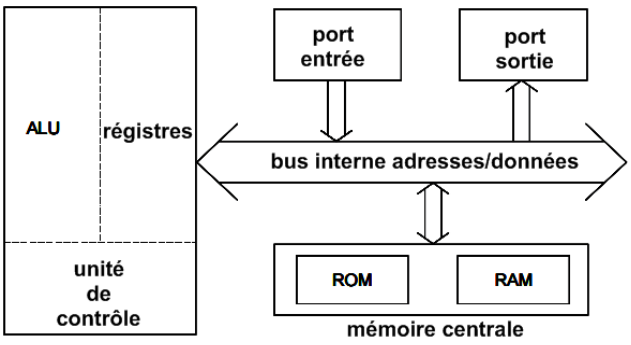
Les périphériques d'entrée permettent d'introduire des données externes dans la CPU, des exemples de ces périphériques sont le clavier et le lecteur CD.

Les périphériques de sortie sont utilisés pour transmettre des données à l'environnement extérieur, généralement les résultats des calculs ; des exemples sont l'écran et l'imprimante.

Mémoire centrale

Elle contient à la fois les instructions de programme et les données utilisées. Elle est matériellement constituée d'une RAM volatile et d'une ROM à lecture seule.

LE MICROPROCESSEUR



Le microprocesseur qui est assimilé à la CPU, lit chaque instruction de la mémoire, la décode et l'exécute. Il traite les données comme indiqué par les instructions. Le traitement consiste en des opérations arithmétiques et logiques traditionnellement réalisées par les circuits combinatoires. Les données représentant les opérandes des opérations proviennent normalement de la mémoire centrale, de même le résultat du traitement est stocké dans la mémoire centrale.

Pour effectuer toutes ces fonctions, le microprocesseur est organisé en différentes unités appropriées. La structure organisationnelle est appelée architecture. Une architecture de microprocesseur typique est illustrée à la Figure en bas de la page2.

Les bus internes

Ces bus permettent l'échange de mots binaires entre les unités. Un bus possède une ligne par bit et l'échange des bits se fait en parallèle.

Trois bus sont utilisés. Le bus d'adresse, le bus de données, et le bus de contrôle

Le bus d'adresse transmet l'adresse vers les emplacements de mémoire ou les périphérique E/S à partir de la CPU.

Cette adresse est reçue par tous les circuits connectés au bus, mais seul le circuit auqel elle est destinée réagit

Le bus de données est utilisé par le CPU pour envoyer et recevoir des données vers et à partir de la mémoire ou les périphériques, y compris les instructions programme stockées dans la mémoire.

Le bus d'adresse est unidirectionnel et le bus de données bidirectionnel.

Le bus de contrôle est utilisé pour transporter des signaux de consigne entre le CPU et divers circuits du système.

Unité arithmétique-logique (ALU)

L'unité arithmétique et logique est un circuit combinatoire qui effectue des opérations arithmétiques et logiques sur les données.

Registres internes

Un certain nombre de registres sont normalement inclus dans le microprocesseur. Ils sont utilisés pour le stockage intermédiaire de données en cours de traitement. Il y a des registres à usage général qui peuvent contenir des données. Il y a d'autres registres qui sont spécialisés :

Accumulateur

C'est un registre utilisé dans les opérations arithmétiques et logiques. Sur les deux opérandes, l'un vient de l'accumulateur (ACC), tandis que l'autre peut être dans un autre registre interne ou peut être amené à partir de la mémoire principale. Le résultat est placé dans l'accumulateur immédiatement après une opération arithmétique ou logique.

Registre d'instructions

Ce registre contient l'instruction à exécuter. Elle est décodée ou interprétée et ce registre fournit en conséquence des signaux à l'unité de contrôle

Compteur de programme PC

C'est un registre qui contient l'adresse de la prochaine instruction à exécuter, qui sera transférée de la mémoire centrale au registre d'instructions. Les instructions de programme stockées dans la mémoire centrale s'exécutent dans l'ordre séquentiel. Normalement ce registre est incrémenté après la lecture de chaque instruction. Cependant, certaines

instructions le modifient de façon que le programme réalise un branchement (saut de programme) ou un appel de sous-programme.

Pointeur de pile SP

C'est un registre utilisé pour pointer une pile, c'est-à-dire une mémoire FILO contenant les adresses de retour d'un sous-programme.

Registre d'état de programme PSW

Un registre d'état est constitué des flags utilisés comme indicateurs pour les instructions conditionnelles. Les flags les plus communs sont :

Le flag **Z** 'zéro' est mis à 1 si le résultat de l'instruction est 0.

Le flag **S** 'Signe' définit si le MSB du résultat est 1.

Le flag **P** 'Parité' indique si le nombre de bits dans le résultat est pair.

Le flag **C** 'Carry' indique si une opération arithmétique donne lieu à un report.

Registres dédiés

Il y d'autres registres dans le microprocesseur nécessaires pour son fonctionnement, mais qui ne peuvent pas être manipulés par le programmeur.

Décodeur d'instructions

Il interprète chaque instruction sous la supervision les signaux de commandes issus de l'unité de contrôle.

Unité de contrôle

Elle délivre les activations de transfert de données de tous les registres, et les signaux de consigne à l'unité de traitement.

Elle a comme entrée les signaux provenant du décodeur d'instruction, ainsi que la valeur des flags fournis par l'unité de traitement.

Elle contient l'horloge principale du microprocesseur qui génère le rythme de séquencement de toutes les opérations.

LANGAGE MACHINE

Puisque l'ordinateur stocke et traite les informations sous forme binaire seulement, l'instruction de programme est encodée en binaire, avec la possibilité de l'exprimer en hexadécimal, pas plus. Un programme sous cette forme est représenté en langage machine.

Les instructions machines se présentent sous l'un des formats

- OPCODE
- OPCODE OPERANDE(S)

Le terme opcode désigne un code opératoire qui est appliqué sur un opérande unique ou plusieurs opérandes. Les différents types d'opérandes sont :

DONNEE REGISTRE ADRESSE MEMOIRE

LANGAGE ASSEMBLEUR ASM

Dans un programme en langage assembleur, les instructions et les opérandes sont représentés par des symboles alphanumériques significatifs et faciles à retenir, appelés mnémoniques. Il est relativement plus facile d'écrire et d'analyser un programme en langage assembleur que dans le langage machine. Le programme en langage assembleur est traduit en langage machine pour pouvoir être exécuté.

Le module logiciel qui réalise cette traduction s'appelle aussi Assembleur

JEU D'INSTRUCTIONS

L'ensemble des instructions d'un langage pris en charge par le microprocesseur comprend généralement cinq groupes de types d'instructions :

Instructions de transfert de données

Ces instructions permettent de déplacer des données entre les registres dans le microprocesseur, entre registre et emplacement de mémoire ou entre 2 emplacements mémoire.

Instructions arithmétiques et logiques

Ces instructions permettent d'effectuer l'addition, la soustraction, l'incrémentation ou décrémentation des données dans un registre ou dans la mémoire, ainsi que les opérations logiques bit à bit AND, OR, NOT, XOR, comparaison, décalage et rotation des données dans un registre ou en mémoire

Instruction de saut (branchement)

Les instructions de saut peuvent être conditionnelles ou inconditionnelles, elles incluent aussi les appels de sous-programme et leur instruction de retour, ainsi que les routines d'interruption.

Une instruction conditionnelle est exécutée uniquement si un flag est à un état binaire particulier, par exemple si le résultat de la dernière opération arithmétique est nul.

Instructions diverses :

Généralement il y a des instructions qui ne sont dans aucun des groupes précédents : manipulation de la pile, modification de la valeur des flags, mise en halte du microprocesseur, instruction "pas d'opération", etc…

Exemple de mnémonique assembleur

ADD ACCU, REGB

Pour un microprocesseur dont l'accumulateur est identifié par ACCU et possédant un registre REGB, cette instruction additionne les valeurs contenues dans chaque registre et met la somme dans l'accumulateur.

L'instruction suivante incrémente la valeur du registre REGB d'une unité : **INC REGB**

Chapitre 1 – Généralités sur les processeurs de signaux digitaux DSP

1. EN QUOI UN DSP SE DISTINGUE T'IL D'UN MICROPROCESSEUR?

Bien que fondamentalement apparentés, les DSP sont sensiblement différents des microprocesseurs polyvalents comme le Pentium de Intel. Pour en savoir les raisons, il faut comprendre la nature spécifique des calculs les plus utilisés en traitement du signal. Ce sont essentiellement ces calculs, pratiquement irréalisables au moyen des microprocesseurs, ou très incommodes à implémenter, qui ont stimulé le développement des DSP à partir du milieu des années 1980.

1.1. Un exemple type, le filtre FIR

Considérons une des fonctions les plus courantes en traitement numérique des signaux, le filtrage des signaux, qui consiste à manipuler un signal pour améliorer ses caractéristiques. Le filtrage peut par exemple réduire le bruit, améliorant ainsi le rapport signal sur bruit.

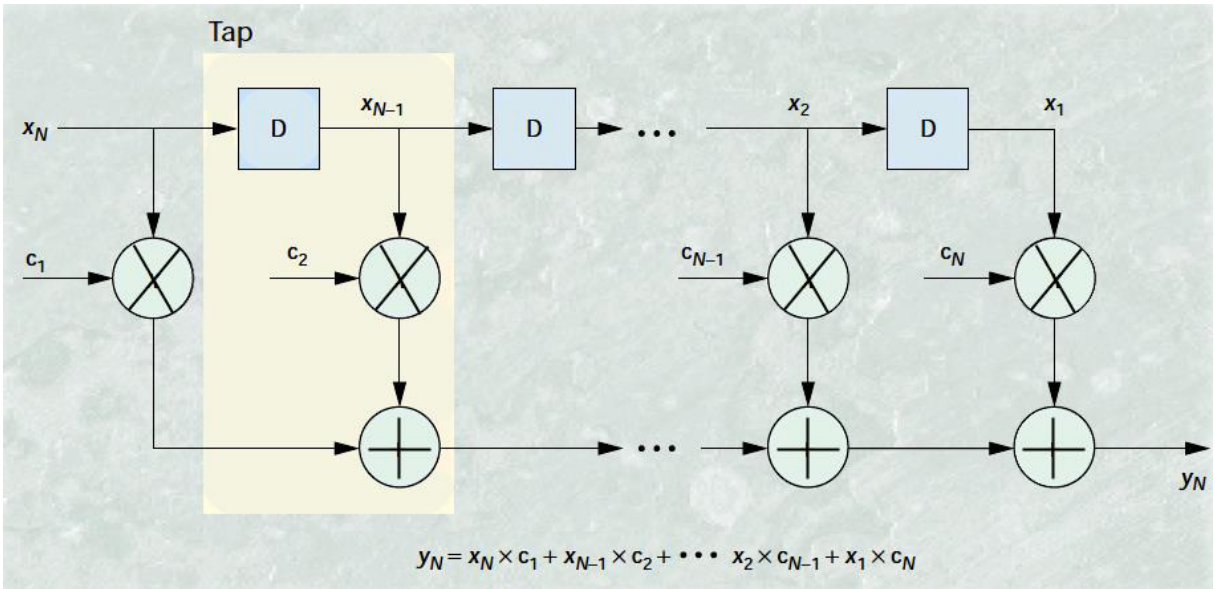
Il peut aussi ne pas paraître évident pourquoi il est souhaitable de filtrer un signal en utilisant un microprocesseur plutôt que des circuits analogiques :

• Un filtre analogique, ou un circuit analogique en général, est sujet à des variations de comportement en fonction des facteurs environnementaux, tels que la température. Un filtre numérique est pratiquement immunisé contre de tels effets.

• Un filtre numérique est facilement reproductible avec des tolérances très strictes, car ses caractéristiques globales sont en général dissociées des caractéristiques individuelles de ses composants élémentaires, pouvant tous dévier différemment de leur comportement nominal.

• Une fois fabriqué, les caractéristiques d'un filtre analogique (comme sa bande passante) sont très difficilement modifiables. Les caractéristiques d'un filtre numérique à base de microprocesseur, peuvent être modifiées simplement par programmation.

Il existe plusieurs types de filtres numériques. Un type couramment utilisé est appelé filtre à réponse impulsionnelle finie (FIR), représenté sur la figure suivante.



L'algorithme de filtrage FIR est assez simple. Les blocs marqués D sont des opérateurs de retard unité; Leur sortie est une copie de l'entrée, retardée d'une période d'échantillonnage. L'ensemble de ces blocs D est désigné par ligne à retard. Dans un système à base de microprocesseur, une série d'emplacements mémoire est habituellement utilisée pour implémenter une ligne à retard.

A tout instant, les N–1 échantillons d'entrée reçus les plus récents sont présents dans la ligne à retard, N étant le nombre total d'échantillons d'entrée utilisés dans le calcul d'un échantillon de sortie y_N. Les échantillons d'entrée sont désignés par x_k; Le premier échantillon d'entrée est x_1, le suivant est x_2, et ainsi de suite.

Chaque fois qu'un nouvel échantillon d'entrée arrive, l'opération de filtrage FIR transfère les échantillons préalablement stockés d'une position vers la droite le long de la ligne à retard. L'échantillon d'entrée le plus ancien est perdu. Un nouvel échantillon de sortie est ensuite calculé en multipliant l'échantillon nouvellement arrivé et chacun des échantillons d'entrée stockés précédemment par un coefficient correspondant. Sur la figure, les coefficients sont représentés par c_k, où k est l'indice du coefficient. La somme des produits de multiplication forme le nouvel échantillon de sortie, y_N.

La combinaison d'un élément de retard, de l'opération de multiplication associée et de l'opération d'addition associée, s'appelle un *tap* (robinet). C'est le cadre à fond clair sur la figure. Le nombre de taps et les valeurs choisies pour les coefficients définissent entièrement le filtre FIR. Par exemple, si les valeurs des coefficients sont toutes égales à l'inverse du nombre de taps, 1/N, la sortie y_N est égale à la moyenne arithmétique des N derniers échantillons reçus, appelée en traitement de signal *moyenne glissante*, qui est en pratique un filtrage passe-bas. Plus généralement, les coefficients c_k sont déterminés en fonction de la réponse en fréquence désirée pour le filtre à concevoir.

Le filtre FIR effectue un produit entre les coefficients et une fenêtre glissante d'échantillons d'entrée x, et cumule les résultats de toutes les multiplications pour former un échantillon de sortie. Mathématiquement, il s'agit d'un produit scalaire de deux vecteurs, connu dans la terminologie DSP par opération MULTIPLY-ACCUMULATE (MAC). C'est l'opération la plus distinctive des DSP.

En vérité ce type d'opération est présent dans beaucoup d'autres applications en traitement de signal, comme la FFT, la corrélation, etc…

Pour implémenter efficacement une opération MAC, un processeur doit d'abord être cabale d'effectuer une multiplication performante. Les microprocesseurs n'étaient pas conçus pour des tâches à multiplications intensives. Ils nécessitent en général des instructions à plusieurs cycles d'horloge pour effectuer une seule multiplication. La première grande modification architecturale qui a distingué les DSP des microprocesseurs a été l'implémentation dans l'unité arithmétique et logique de circuits spécialisés permettant la multiplication en un seul cycle d'horloge.

Notez aussi que les registres nécessaires pour contenir la somme de plusieurs produits doivent être de plus grande taille que les registres des opérandes, donnant lieu à un type particulier d'accumulateur dans les DSP. Enfin, pour tirer parti des circuits multiply-accumulate spécialisés, les jeux d'instructions des DSP incluent toujours une instruction MAC explicite.

Ce circuit MAC combiné a une instruction MAC spécialisée ont été les deux principaux facteurs de distinction entre les premiers DSP et les microprocesseurs.

1.2. Organisation de la mémoire

Revenons à l'exemple du filtre FIR. Il est convenu que le processeur dispose d'un circuit MAC qui permettrait qu'un étage du filtre FIR s'exécute en un cycle d'horloge. Mais qu'en est-il des accès mémoire liés à cette opération. Le processeur doit

• lire en mémoire et décoder l'instruction MAC,

• lire en mémoire la valeur de l'échantillon x_k,

• lire la valeur du coefficient approprié,

• écrire la valeur de l'échantillon à l'emplacement mémoire suivant, afin de décaler les données de la ligne à retard.

Le processeur doit effectuer au total quatre accès mémoire pour un cycle d'instruction.

Dans une architecture Von Neumann, quatre accès mémoire consommeraient au minimum quatre cycles d'instructions. Même si le processeur inclue le circuit arithmétique nécessaire pour réaliser des MAC à un cycle, il ne peut pas concrètement réaliser l'objectif d'un cycle par étage. Rien que pour cette raison, les processeurs DSP utilisent une architecture Harvard.

Dans l'architecture Harvard, il existe deux mémoires séparées, une pour les données et une pour les instructions du programme, et deux bus séparés qui les relient au processeur. Cela permet d'accéder aux instructions du programme et aux données en même temps.

Dans la plupart des DSP commerciaux, l'architecture Harvard est améliorée en ajoutant une petite mémoire interne rapide, appelée «cache d'instructions», dans laquelle sont relocalisées dynamiquement au moment de l'exécution les instructions du programme les plus fréquemment (ou les plus récemment) exécutées.

Ceci est très avantageux par exemple si le DSP exécute une boucle suffisamment petite pour que toutes ses instructions puissent être contenues dans le cache d'instructions. Ces instructions sont transférées dans le cache d'instructions lors de l'exécution de la première itération de la boucle. Les autres itérations sont exécutées directement à partir du cache d'instructions.

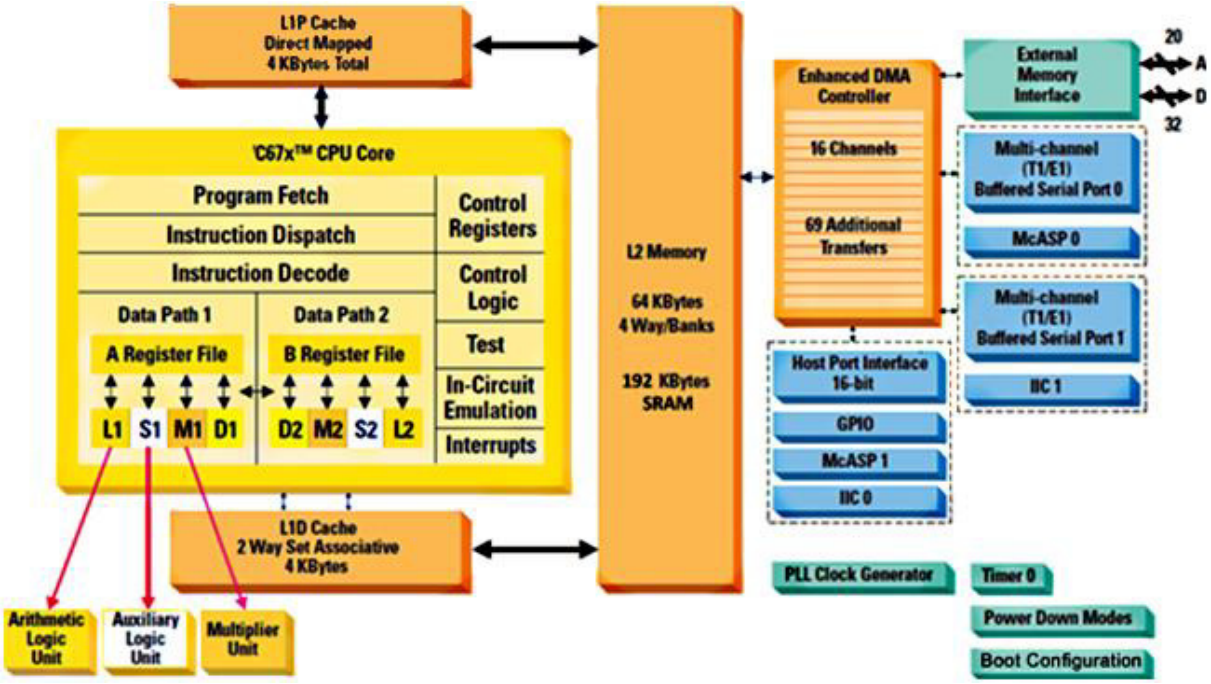
Une autre amélioration plus récente de l'architecture Harvard comme celle du DSP de Texas Instruments TMS320C6713, inclut un cache de programme et un cache de données. De plus Il existe deux niveaux de cache internes, appelés Niveau 1 **L1** et Niveau 2 **L2**.

Le temps d'accès typique est de l'ordre de 1ns pour le cache L1, et 5ns pour le cache L2

2. Le TMS320C6713 : Architecture

Le DSP de Texas Instruments TMS320C6713 est réalisé en technologie CMOS. Il utilise l'arithmétique à virgule flottante, et une architecture **VLIW** Very Long Instruction Word.

La figure suivante illustre l'intérieur du boitier du TMS320C6713.



La mémoire interne inclue deux niveaux de cache. Le cache L1 comprend 8 koctets de mémoire divisés en 4 koctets de cache de programme L1P et 4 koctets de cache de données L1D. Le cache L2 comprend 256 koctets de mémoire divisée en une mémoire SRAM de 192 koctets, et une mémoire à accès-dual de 64 koctets.

La mémoire interne de programme est structurée de sorte que le pipeline Fetch-Dispatch peut envoyer huit instructions parallèles au décodeur d'instructions à chaque cycle.

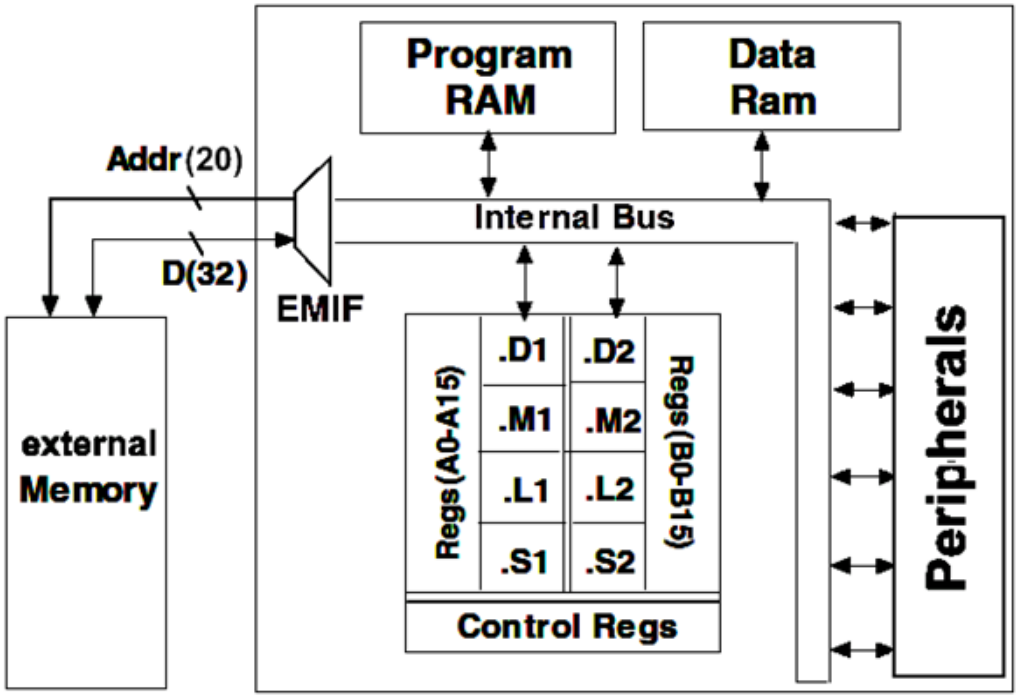
L'unité de traitement centrale du TMS320C6713 est divisée en deux sous-ensembles appelés chemin de données **A** et chemin de données **B.**

Outre ces deux chemins de données, l'unité centrale comporte une logique de contrôle et des registres spécialisés nécessaires à son fonctionnement.

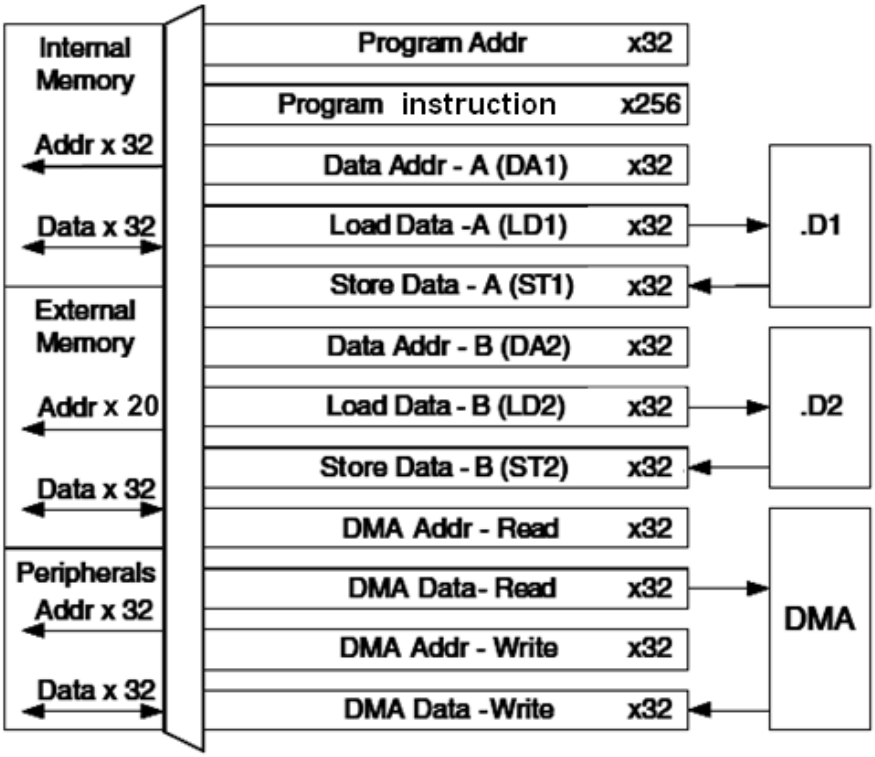
Chacun des deux chemins de données comporte quatre unités fonctionnelles autonomes, qui peuvent exécuter chacune une instruction séparée, et qui dispose de son jeu d'instructions dédié.

- une unité **.M** utilisée pour l'opération de multiplication
- une unité **.L** utilisée pour les opérations logiques et arithmétiques
- une unité **.S** utilisée pour les branchements et boucles du programme, les décalages de registres, la manipulation de bits, et les opérations arithmétiques
- une unité **.D** utilisée pour la lecture et l'écriture dans la mémoire de données, et pour les opérations arithmétiques

Il ya 16 registres 32 bits associés à chaque chemin de donnée, **A0-A15** côté **A**, **B0-B15** coté **B**. Toute interaction avec les unités fonctionnelles se fait obligatoirement à travers ces registres.



Le bus interne est constitué d'un bus 32 bits d'adresse de programme, un bus 256 bits d'instructions de programme pouvant recevoir huit instructions 32 bits, deux bus 32 bits d'adresses de données **DA1** et **DA2**, deux bus 32 bits chargement de données **LD1** et **LD2**, et deux bus 32 bits stockage de données **ST1** et **ST2**. En outre, il existe un bus 32 bits de données DMA et un bus 32 bits d'adresses DMA. La mémoire hors-puce ou externe est accessible via un bus 20 bits d'adresses et un bus 32 bits de données contrôlés par EMIF External Memory Interface.



Le concept **VLIW Very Long Instruction Word** a été introduit pour la première fois par TI pour cette famille de DSP, il désigne des codes instructions de taille fixe 256 bits, divisés en huit mots de 32 bits destiné chacun en parallèle à l'une des huit unités fonctionnelles, de sorte que idéalement les huit instructions devraient s'exécuter simultanément pour une pleine exploitation du parallélisme, en pratique cet optimum est rarement réalisé.

Les périphériques

Les périphériques disponibles dans le TMS320C6713 sont répartis en deux catégories : Les interconnexions entrées-sorties, et les services système

Interconnexions :

EMIF, External Memory Interface, fournit les signaux nécessaires pour contrôler les accès à la mémoire externe SDRAM, SBSRAM, SRAM, ROM/Flash, FPGA

DMA, Direct Memory Access permet le déplacement des données d'un emplacement en mémoire vers un autre sans intervention de l'Unité Centrale

McBSP Multichannel Buffered Serial Port : 128 canaux de communication full-duplex programmables, à double registres tampons de données qui permettent un flux continu indépendant en réception et en transmission. Directement interfaçable avec les codecs TDM haut débit, l'interface analogique AIC, l'interface sérielle SPI, les Codecs AC97, et EEPROM sérielle

McASP Multichannel Audio Serial Port : port série polyvalent optimisé pour les applications audio multicanaux. Inclue le flux multiplexé par répartition dans le temps TDM, les protocoles Inter Integrated Sound I2S à ADC multicanal et Digital audio Interface Transmission DIT à sortie multiple, DAC, Codec, et DIR Digital Infrared

HPI Host Port Interface, permet à un autre circuit (hôte) d'accéder à la mémoire interne

IIC, bus série standard Inter-Integrated Circuit

GPIO, interface parallèle universelle General Purpose Input Output.

Services Système :

Timer, temporisateur à usage général constitué de deux compteurs 32 bits.

Contrôleur PLL, assure la synchronisation d'horloge pour les périphériques à partir de signaux internes ou externes.

Power Down, unité de mise hors tension pour économiser de l'énergie lorsque la CPU est inactive, ou de réduire la fréquence des signaux si besoin est car la dissipation de la puissance des circuits CMOS se produit lors du basculement d'un état logique à l'état inverse.

Boot Configuration, permet la programmation par l'utilisateur du mode d'amorçage, à partir de la mémoire hors-puce, ou autre

Avant d'illustrer par l'exemple le mode de fonctionnement de l'unité de traitement, Il est utile de présenter à ce niveau de l'exposé une vue d'ensemble du jeu d'instructions du C6713 qui sera étudié en détail plus tard. Chaque unité fonctionnelle à ses instructions propres, à part celles écrites en bleu qui sont communes à plus d'une unité

.S Unit	
ADD	MVKLH
ADDK	NEG
ADD2	NOT
AND	OR
B	SET
CLR	SHL
EXT	SHR
MV	SSHL
MVC	SUB
MVK	SUB2
MVKL	XOR
MVKH	ZERO

.L Unit	
ABS	NOT
ADD	OR
AND	SADD
CMPEQ	SAT
CMPGT	SSUB
CMPLT	SUB
LMBD	SUBC
MV	XOR
NEG	ZERO
NORM	

.M Unit	
MPY	SMPY
MPYH	SMPYH

.D Unit	
ADD	STB/H/W
ADDA	SUB
LDB/H/W	SUBA
MV	ZERO
NEG	

No Unit	
NOP	IDLE

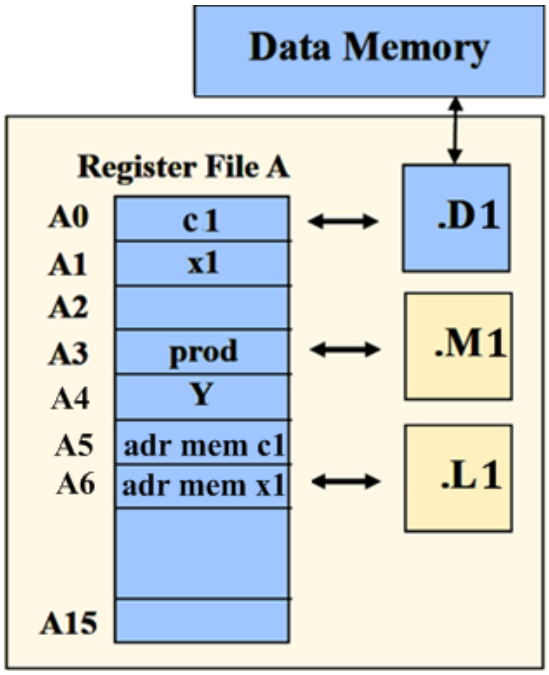
L'exemple montre comment l'unité de traitement centrale est mise en œuvre et pourrait être utilisée pour effectuer l'opération **MAC** du filtre FIR $Y = \sum_{n=1}^{10} c_n x_n$

En pratique les coefficients c_n sont préprogrammés et disponibles dans la mémoire de données, les échantillons x_k sont dans une autre zone mémoire après acquisition via une interface hardware.

l'unité **.D1** effectue le chargement des opérandes $\mathbf{c_n}$, $\mathbf{x_n}$ dans les registres à partir de la mémoire en utilisant une instruction de chargement telle que **LDH .D1 *R_n,R_m** qui signifie que l'adresse mémoire pointée par $\mathbf{R_n}$ est chargé dans le registre $\mathbf{R_m}$.

L'unité **.M1** effectue les multiplications de façon câblée entre les variables c_n et x_n, et met le résultat dans une variable **prod,** instruction assembleur **MPY .M1 c1, x1, prod**

L'unité **.L1** effectue l'addition, résultat cumulé dans **Y**, instruction **ADD .L1 Y,prod,Y**



$$Y = \sum_{n=1}^{10} c_n x_n$$

```
LDH  .D1  *A5,A0
LDH  .D1  *A6,A1
MPY  .M1  A0,A1,A3
ADD  .L1  A4,A3,A4
```

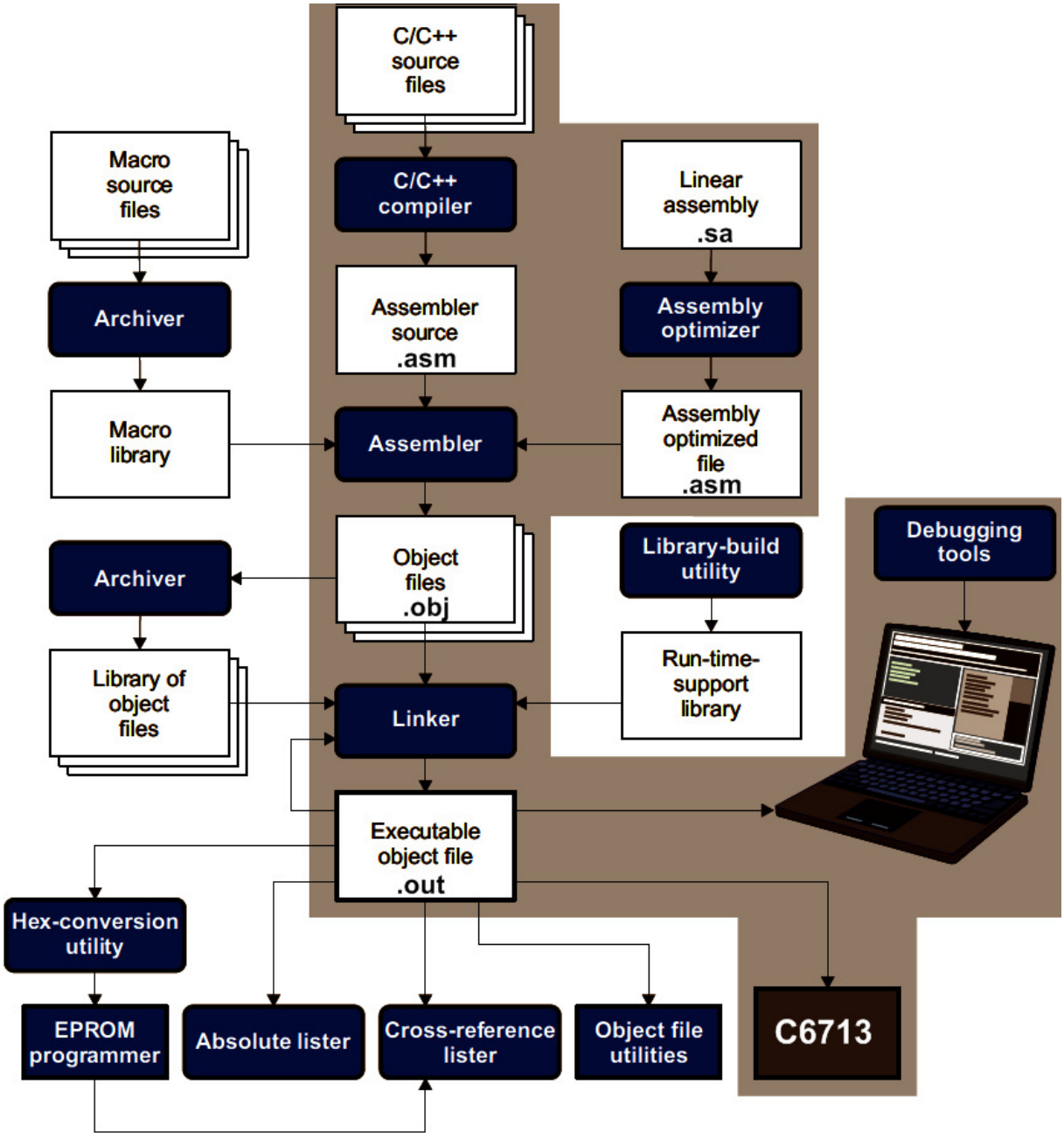
2. Mappage Mémoire TMS320C6713

MEMORY BLOCK DESCRIPTION	BLOCK SIZE	HEX ADDRESS RANGE
Internal RAM (L2)	192K	0000 0000 – 0002 FFFF
Internal RAM/Cache	64K	0003 0000 – 0003 FFFF
Reserved	24M – 256K	0004 0000 – 017F FFFF
External Memory Interface (EMIF) Registers	256K	0180 0000 – 0183 FFFF
L2 Registers	128K	0184 0000 – 0185 FFFF
Reserved	128K	0186 0000 – 0187 FFFF
HPI Registers	256K	0188 0000 – 018B FFFF
McBSP 0 Registers	256K	018C 0000 – 018F FFFF
McBSP 1 Registers	256K	0190 0000 – 0193 FFFF
Timer 0 Registers	256K	0194 0000 – 0197 FFFF
Timer 1 Registers	256K	0198 0000 – 019B FFFF
Interrupt Selector Registers	512	019C 0000 – 019C 01FF
Device Configuration Registers	4	019C 0200 – 019C 0203
Reserved	256K − 516	019C 0204 – 019F FFFF
EDMA RAM and EDMA Registers	256K	01A0 0000 – 01A3 FFFF
Reserved	768K	01A4 0000 – 01AF FFFF
GPIO Registers	16K	01B0 0000 – 01B0 3FFF
Reserved	240K	01B0 4000 – 01B3 FFFF
I2C0 Registers	16K	01B4 0000 – 01B4 3FFF
I2C1 Registers	16K	01B4 4000 – 01B4 7FFF
Reserved	16K	01B4 8000 – 01B4 BFFF
McASP0 Registers	16K	01B4 C000 – 01B4 FFFF
McASP1 Registers	16K	01B5 0000 – 01B5 3FFF
Reserved	160K	01B5 4000 – 01B7 BFFF
PLL Registers	8K	01B7 C000 – 01B7 DFFF
Reserved	264K	01B7 E000 – 01BB FFFF
Emulation Registers	256K	01BC 0000 – 01BF FFFF
Reserved	4M	01C0 0000 – 01FF FFFF
QDMA Registers	52	0200 0000 – 0200 0033
Reserved	16M − 52	0200 0034 – 02FF FFFF
Reserved	720M	0300 0000 – 2FFF FFFF
McBSP0 Data Port	64M	3000 0000 – 33FF FFFF
McBSP1 Data Port	64M	3400 0000 – 37FF FFFF
Reserved	64M	3800 0000 – 3BFF FFFF
McASP0 Data Port	1M	3C00 0000 – 3C0F FFFF
McASP1 Data Port	1M	3C10 0000 – 3C1F FFFF
Reserved	1G + 62M	3C20 0000 – 7FFF FFFF
EMIF CE0†	256M	8000 0000 – 8FFF FFFF
EMIF CE1†	256M	9000 0000 – 9FFF FFFF
EMIF CE2†	256M	A000 0000 – AFFF FFFF
EMIF CE3†	256M	B000 0000 – BFFF FFFF
Reserved	1G	C000 0000 – FFFF FFFF

† The number of EMIF address pins (EA[21:2]) limits the maximum addressable memory (SDRAM) to 128MB per CE space.

Chapitre 2 – Programmation du TMS320C6713

Le DSP TMS302C6713 peut être programmé en langage C ou en langage assembleur. La figure ci-dessous montre le flot de conception logicielle. La partie sur fond sombre montre la voie la plus couramment suivie pour passer d'un fichier ***source*** : extension **.c** pour C, .**asm** pour assembleur, ou **.sa** pour assemblage linéaire, vers un fichier ***exécutable*** : extension **.out**



Les services assemblage, édition de liens, compilation, et débogage sont fournis dans l'environnement de développement intégré **Code Composer Studio CCS**.

Le compilateur C6713 offre un support de haut niveau qui permet de transformer un code C/C++ en un code source en langage assembleur plus efficace. Les outils de compilation sont inclus dans un programme **cl6x**, qui est utilisé pour compiler, optimiser l'assemblage, assembler, et lier les programmes en une seule opération.

Bien que l'écriture de programmes en C nécessite moins d'efforts, l'efficacité obtenue est normalement inférieure à celle des programmes écrits en assembleur. L'efficacité signifie avoir aussi peu d'instructions ou de cycles d'instruction que possible en utilisant au maximum les ressources de la puce DSP.

En pratique, on écrit un programme C pour analyser le comportement et la fonctionnalité d'un algorithme. Ensuite, si la vitesse de traitement n'est pas satisfaisante en utilisant le compilateur C optimisé, les parties lentes du code C sont identifiées et converties en assembleur. De même, le code tout entier peut être réécrit en assembleur.

En plus du C et de l'assembleur manuel, le C6713 possède un optimiseur d'assemblage appelé assembleur linéaire.

Le code écrit pour l'optimiseur d'assemblage est similaire au code source assembleur, sauf qu'il ne comprend pas d'informations sur les instructions parallèles, les latences du pipeline d'instructions, ainsi que l'affectation des registres. L'optimiseur d'assemblage prend soin de rationaliser le code en détectant des instructions qui peuvent être exécutées en parallèle, en gérant les latences de pipeline et en prenant en charge l'affectation des registres et l'unité à utiliser

L'optimiseur d'assemblage et le compilateur sont utilisés pour convertir, respectivement, un fichier assembleur linéaire ou un fichier C en un fichier **.asm**

L'assembleur est utilisé pour convertir un fichier assembleur **.asm** en un fichier ***objet*** (extension **.obj**).

L'éditeur de liens est utilisé pour combiner les fichiers objet en un fichier exécutable, selon les instructions indiquées dans le fichier de commande de l'éditeur de liens **.cmd**

Structure du code assembleur

Un programme en langage assembleur doit être un fichier texte ASCII.
Toute ligne de code assembleur peut inclure jusqu'à sept éléments selon le format :

`Label: || [condition] instruction unit operands ; comment`

`Label:` Étiquette, jusqu'à 32 caractères alphanumériques, le premier caractère est positionné sur la première colonne du fichier texte, généralement c'est le caractère underscore _

`||` **Barres parallèles**, exécution en parallèle

`[condition]` Si une condition est spécifiée l'instruction s'exécute dans le cas où cette condition est vraie. Si elle n'est pas spécifiée, l'instruction est toujours exécutée.

`Instruction` c'est soit une directive, soit des mnémoniques:

Les directives d'assemblage sont des commandes **asm6x** qui contrôlent le processus d'assemblage ou définissent les structures de données (constantes et variables). Les directives d'assemblage commencent par un point. Les mnémoniques du processeur sont les instructions qui s'exécutent dans le programme. Les mnémoniques doivent commencer à la colonne 2 ou plus.

`unit` Unité fonctionnelle. La spécification de l'unité fonctionnelle dans le code assembleur est optionnelle, pour documenter le code sur la ressource utilisée par l'instruction

`operands` Les instructions C6713 utilisent trois types d'opérandes pour accéder aux données :

les registre, les constants, et les pointeurs d'adresses de données.

Seules les instructions de chargement et de stockage de données en mémoire utilisent des opérandes pointeur

Toutes les instructions C6713 ont un opérande destination. La plupart des instructions nécessitent un ou deux opérandes source, qui doivent être dans le même chemin de données.

Un opérande source par paquet d'instructions parallèles peut provenir du chemin opposé, dans ce cas on ajoute X au nom de l'unité pour *chemin croisé*.

`;comment` commentaires qui aident la lisibilité du programme, sans effet sur son exécution

Exemple de code assembleur Produit scalaire $y = \sum_{n=1}^{10} c_n x_n$

vecteurs c et x à 10 composantes de type entiers signés

```
Label   Cond.   Instr.   unit   Operands        Comment
                MVK      .S1    c,A5        ;move address of c
                MVKH     .S1    c,A5        ;into register A5
                MVK      .S1    x,A6        ;move address of x
                MVKH     .S1    x,A6        ;into register A6
                MVK      .S1    y,A7        ;move address of y
                MVKH     .S1    y,A7        ;into register A7
                MVK      .S1    10,A2       ;A2=10, loop counter
loop:           LDH      .D1    *A5++,A0    ;A0=cn
                LDH      .D1    *A6++,A1    ;A1=xn
                MPY      .M1    A0,A1,A3    ;A3=cn*xn, product
                ADD      .L1    A3,A4,A4    ;y=y+A3
                SUB      .L1    A2,1,A2     ;decr loop counter
        [A2]    B        .S1    loop        ;if A2≠0 branch to loop
                STH      .D1    A4,*A7      ;*A7=y
```

Il convient de mentionner que l'assembleur n'est pas case-sensitive, c'est-à-dire, les instructions et les registres peuvent être écrits indifféremment en minuscules ou en majuscules.

Dans cet exemple, seules les unités fonctionnelles A sont utilisées, et 8 parmi les 16 les registres sont affectés :

A0	xn
A1	cn
A2	compteur de boucle
A3	produit cn*xn

A4	y
A5	adresse de cn
A6	adresse de xn
A7	adresse de y

Un compteur de boucle est configuré en utilisant l'instruction de déplacement de constante **MVK**. Cette instruction utilise l'unité **.S1** pour placer la constante 10 décimal dans le registre A2.
Le début de la boucle est indiqué par l'étiquette **loop**
La fin de boucle est indiquée une instruction de soustraction **SUB** qui décrémente le compteur de boucle A2 suivi d'une instruction de branchement **B** ver le début de la boucle si A2 n'est pas nul.

Les crochets **[ ]** dans l'instruction de branchement indiquent qu'il s'agit d'une instruction conditionnelle.
Les instructions du C6713 peuvent être conditionnées en fonction d'une valeur nulle ou non nulle dans l'un des registres : **A1, A2, B0, B1** et **B2**.
La syntaxe **[A2]** signifie "exécuter l'instruction si A2≠0", **[! A2]** signifie "exécuter l'instruction si A2=0".

MVK et **MVKH** sont utilisés pour charger l'adresse de cn, de xn, et de y, respectivement dans les registres A5, A6 et A7. Ces instructions doivent être exécutées dans l'ordre indiqué pour charger d'abord les 16 bits inférieurs de l'adresse 32 bits complète, suivis des 16 bits supérieurs. C'est à dire l'instruction MVK écrase le 16-MSB du registre cible.

Les registres A5, A6, A7, sont utilisés comme pointeurs pour charger cn, xn dans les registres A0, A1 et stocker y à partir du registre A4 après la fin de la boucle.
Selon le type de données, on peut utiliser l'une des instructions de chargement suivantes: **LDB** octets (8 bits), **LDH** demi-mots (half-word 16 bits), ou **LDW** mots (word 32 bits)**.** Dans notre exemple, les données sont supposées être de 16 bit.
Notez que les pointeurs A5 et A6 doivent être post-incrémentés, de sorte qu'ils pointent l'adresse suivante à la prochaine itération de la boucle.

Les instructions **MPY** et **ADD** dans la boucle exécutent l'opération de produit scalaire. L'instruction MPY est effectuée par l'unité .M1 et ADD par l'unité .L1.

Enfin, il convient de mentionner que le code ci-dessus tel quel ne fonctionnera pas correctement sur le C6713 à cause du pipeline des instructions

Sur le processeur C6713, la lecture d'une instruction se fait en quatre phases, chacune nécessitant un cycle. Celles-ci incluent générer l'adresse de l'instruction, envoyer l'adresse à la mémoire, attendre l'opcode, et lire l'opcode de la mémoire.

Le décodage de l'instruction se fait en deux phases, chacune nécessitant un cycle d'horloge. Ce sont l'envoi aux unités fonctionnelles appropriées (dispatch), et le décodage par l'unité.

L'exécution varie de une à six phases selon les instructions. Ce qui peut donner lieu à un maximum de 5 retards. En raison des retards associés aux instructions de multiplication (MPY 1 retard), de chargement (LDH 4 retards) et de branchement (B 5 retards), un nombre approprié de **NOP** (no operation ou délai) doit être inséré lorsque le résultat d'une instruction est utilisé par l'instruction suivante dans un programme afin que le pipeline fonctionne correctement.

Par conséquent, pour que l'exemple précédent s'exécute sur le processeur C6713, un NOP après la multiplication MPY, 4 NOP après le chargement LDH, 5 NOP après le branchement B, doivent être insérés.

```
loop:       LDH  .D1  *A5++,A0  ;A0=cn
            LDH  .D1  *A6++,A1  ;A1=xn
            NOP       4
            MPY  .M1  A0,A1,A3  ;A3=cn*xn, product
            NOP
            ADD  .L1  A3,A4,A4  ;y=y+A3
            SUB  .L1  A2,1,A2   ;decr loop counter
      [A2]  B    .S1  loop      ;if A2≠0 branch to loop
            NOP       5
            STH  .D1  A4,*A7    ;*A7=y
```

Pipeline des instructions

Dans tout processeur, le traitement d'une instruction de programme se déroule sur trois étapes : lecture mémoire, décodage, exécution. Sans le mécanisme de pipeline ces trois étapes doivent être entièrement achevées pour une instruction, afin que le traitement de l'instruction suivante dans le programme puisse s'entamer. Le DSP C6713 dispose d'une circuiterie qui permet de dissocier dans une certaine mesure ces étapes, et permet un chevauchement dans le temps entre instructions, d'où un gain de temps appréciable.

1. Dans le DSP C6713 la lecture de l'instruction en mémoire se compose de quatre phases pour toutes les instructions, chaque phase consomme un cycle d'horloge :
PG (generate) : génère de l'adresse du programme dans la CPU pour lire l'opcode
PS (send) : envoi de l'adresse sur le bus adressemémoire de programme
PW (wait) : attente de l'opcode sur le bus de données mémoire de programme
PR (read) : réception à la CPU du paquet opcode à partir de la mémoire

2. Le décodage de l'instruction se compose de deux phases pour toutes les instructions :
DP : dispatch des instructions aux unités fonctionnelles appropriées
DC : décodage des instructions

3. L'étape d'exécution dans l'unité fonctionnelle varie de 1 phase à 6 phases maximum en arithmétique à virgule fixe, et à 10 phases maximum en arithmétique à virgule flottante en fonction des retards (latences) liés à chaque instruction spécifique.

Lecture opcode				Décodage		Execution (virgule flottante)									
PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10

A tout instant 16 instructions successives du programme peuvent occuper les trois étages du pipeline

Cycle#	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
LDH	PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5					
MPY		PG	PS	PW	PR	DP	DC	E1	E2							
ADD			PG	PS	PW	PR	DP	DC	E1							
SUB				PG	PS	PW	PR	DP	DC	E1						
B					PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5	E6
STH						PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5

Les colonnes du tableau représentent les cycles de l'horloge 1, 2, 3,….
Les lignes représentent les phases du pipeline pour 6 instructions du programme

La plupart des instructions ont un seul cycle d'exécution. Les instructions telles que la multiplication **MPY**, le chargement **LDH,** et le branchement **B** prennent respectivement 2, 5, et 6 cycles en virgule fixe.
Par conséquent, pour que l'exemple s'exécute correctement sur le DSP C6713, un NOP doit être inséré après MPY, 4 NOP après le deuxième LDH, et 5 NOP après le branchement B.

.D .L .S units shared instructions

ADD (.unit) *src1*, *src2*, *dst*	*Add Two Signed Integers Without Saturation*
MV (.unit) *src2, dst*	*Move From Register to Register*
SUB (.unit) *src1*, *src2*, *dst*	*Subtract Two Signed Integers Without Saturation*
ZERO (.unit) *dst*	*Zero a Register*

.D unit instructions

ADDAB/D/H/W (.unit) *src2*, *src1*, *dst*	*Add Using B/D/H/W Addressing*
LDB/H/W/DW (.unit) **+baseR[offsetR], dst* **LDB/H/W/DW** (.unit) **+baseR[ucst5], dst*	*Load B/H/W/DW From Memory With a Register Offset or 5-Bit Unsigned Constant Offset*
STB/H/W (.unit) *src, *+baseR[offsetR]* **STB/H/W** (.unit) *src, *+baseR[ucst5]*	*Store B/H/W to Memory With a Register Offset or 5-Bit Unsigned Constant Offset*
SUBAB/H/W (.unit) *src2*, *src1*, *dst*	*Subtract Using B/H/W Addressing Mode*

.L .S units shared instructions

ADDDP/SP (.unit) *src1*, *src2*, *dst*	*Add Two Double/Simple-Precision Floating-Point Values*
AND (.unit) *src1*, *src2*, *dst*	*Bitwise AND*
NEG (.unit) *src2, dst*	*Negate*
NOT (.unit) *src2, dst*	*Bitwise NOT*
OR (.unit) *src1*, *src2*, *dst*	*Bitwise OR*
SUBDP/SP (.unit) *src1*, *src2*, *dst*	*Subtract Two Double/Simple-Precision Floating-Point Values*
SUBU (.unit) *src1*, *src2*, *dst*	*Subtract Two Unsigned Integers Without Saturation*
XOR (.unit) *src1*, *src2*, *dst*	*Bitwise Exclusive OR*

.L unit instructions

ABS (.unit) *src2*, *dst*	*Absolute Value With Saturation*
ADDU (.unit) *src1*, *src2*, *dst*	*Add Two Unsigned Integers Without Saturation*
CMPEQ (.unit) *src1*, *src2*, *dst*	*Compare for Equality, Signed Integers*
CMPGT (.unit) *src1*, *src2*, *dst*	*Compare for Greater Than, Signed Integers*
CMPGTU (.unit) *src1*, *src2*, *dst*	*Compare for Greater Than, Unsigned Integers*
CMPLT (.unit) *src1*, *src2*, *dst*	*Compare for Less Than, Signed Integers*
CMPLTU (.unit) *src1*, *src2*, *dst*	*Compare for Less Than, Unsigned Integers*
SADD (.unit) *src1*, *src2*, *dst*	*Add Two Signed Integers With Saturation*
SAT (.unit) *src2*, *dst*	*Saturate a 40-Bit Integer to a 32-Bit Integer*
SSUB (.unit) *src1*, *src2*, *dst*	*Subtract Two Signed Integers With Saturation*

.S unit instructions

ABSDP/SP (.unit) *src2*, *dst* *Absolute Value, Double/Single-Precision Floating-Point*

ADDK (.unit) *cst*, *dst* *Add Signed 16-Bit Constant to Register*

ADD2 (.unit) *src1*, *src2*, *dst* *Add Two 16-Bit Integers on Upper and Lower Register Halves*

B (.unit) label *Branch Using a Displacement*

B (.S2) *src2* *Branch Using a register*

B (.S2) **IRP** *Branch Using an Interrupt Return Pointer*

B (.S2) **NRP** *Branch Using NMI Return Pointer*

CLR (.unit) *src2*, *csta*, *cstb*, *dst* or **CLR** (.unit) *src2*, *src1*, *dst* *Clear a Bit Field*

CMPGTDP/SP (.unit) *src1*, *src2*, *dst* *Compare for Greater Than, Double/Single-Precision Float.*

CMPLTDP/SP (.unit) *src1*, *src2*, *dst* *Compare for Less Than, Double/Single -Precision Float.*

EXT (.unit) *src2*, *csta*, *cstb*, *dst* or **EXT** (.unit) *src2*, *src1*, *dst* *Extract and Sign-Extend a Bit Field*

EXTU (.unit) *src2*, *csta*, *cstb*, *dst* or **EXTU** (.unit) *src2*, *src1*, *dst* *Extract and Zero-Extend a Bit Field*

MVC (.S2) *src2*, *dst* *Move Between Control File and Register File*

MVK (.unit) *cst*, *dst* *Move Signed Constant Into Register and Sign Extend*

MVKH (.unit) *cst*, *dst* *Move 16-Bit Constant Into Upper Bits of Register*

SET (.unit) *src2*, *csta*, *cstb*, *dst* or **SET** (.unit) *src2*, *src1*, *dst* *Set a Bit Field*

SHL (.unit) *src2*, *src1*, *dst* *Arithmetic Shift Left*

SHR (.unit) *src2*, *src1*, *dst* *Arithmetic Shift Right*

SHRU (.unit) *src2*, *src1*, *dst* *Logical Shift Right*

SSHL (.unit) *src2*, *src1*, *dst* *Shift Left With Saturation*

SUB2 (.unit) *src1*, *src2*, *dst* *Subtract Two 16-Bit Integers on Upper and Lower Register Halves*

.M unit instructions

MPY (.unit) *src1*, *src2*, *dst* *Multiply Signed 16 LSB × Signed 16 LSB*

MPYDP/SP (.unit) *src1*, *src2*, *dst* *Multiply Two Double/Single-Precision Floating-Point Values*

MPYI (.unit) *src1*, *src2*, *dst* *Multiply 32-Bit × 32-Bit Into 32-Bit Result*

MPYID (.unit) *src1*, *src2*, *dst* *Multiply 32-Bit × 32-Bit Into 64-Bit Result*

MPYSP (.unit) *src1*, *src2*, *dst* *Multiply Two Single-Precision Floating-Point Values*

MPYSP2DP (.unit) *src1*, *src2*, *dst* *Multiply Two Single-Precision Values, Double-Precision Result*

SMPY (.unit) *src1*, *src2*, *dst* *Multiply Signed 16 LSB × Signed 16 LSB With Left Shift and Saturation*

No unit instructions

IDLE *Multicycle NOP With No Termination Until Interrupt*

NOP [*count*] *No Operation*

Modes d'adressage

Les modes d'adressage mémoire spécifient comment accéder aux données, comme lire un opérande indirectement à partir d'un emplacement mémoire.

Adressage indirect :

Le mode d'adressage indirect peut être linéaire ou circulaire, il utilise un astérisque ***** en conjonction avec l'un des 32 registres 32 bits A0 à A15 et B0 à B15. Ces registres sont alors considérés comme des pointeurs lorsqu'ils figurent comme opérandes dans une instruction :

1. ***R**
   Le registre R contient l'adresse d'un emplacement mémoire où la valeur de la donnée est mémorisée.

2. ***R++(d)**.
   Le registre R contient l'emplacement mémoire. Après l'utilisation de l'adresse, R est post-incrémenté, de sorte que la nouvelle adresse est décalée par la valeur de déplacement d.
   Par défaut d=1.
   Un double moins – – au lieu d'un double plus ++ post-décrémente l'adresse à R–d.

3. *** ++R(d).**
   L'adresse est pré-incrémentée ou décalée par d avant utilisation, de sorte que l'adresse courante soit R+d.
   Un double moins pré-décrémente l'adresse mémoire de sorte que l'adresse utilisée soit R–d.

4. ***+R(d)**
   L'adresse est pré-incrémentée par d, telle que l'adresse courante soit R+d, mais sans mise à jour de R. C'est à dire R revient à sa valeur d'origine après utilisation.
   ***–R(d)** pré-décrémente R sans mise à jour

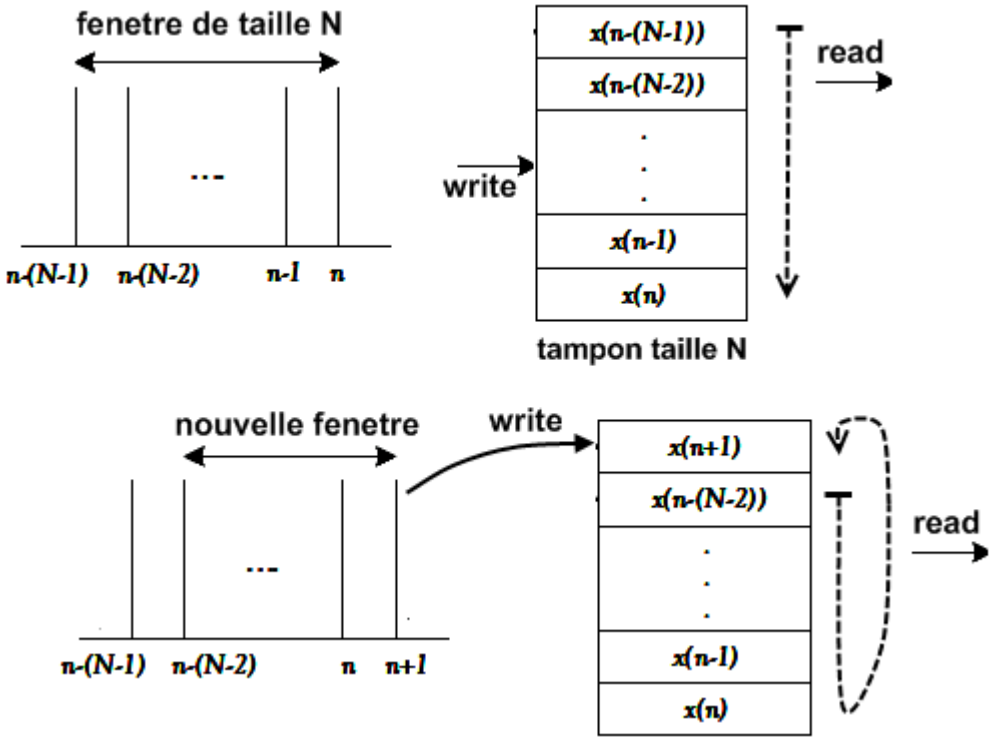
Adressage circulaire :

Ce mode d'adressage utilise un tampon mémoire à accès tournant, généralement de petite taille. Lorsque le pointeur atteint la fin du tampon contenant la dernière donnée et qu'il est incrémenté, il est automatiquement retourné pour pointer le début du tampon qui contient la première donnée.

Ce tampon est matériel, et le C6713 dispose d'un circuit dédié pour permettre ce mode d'adressage.

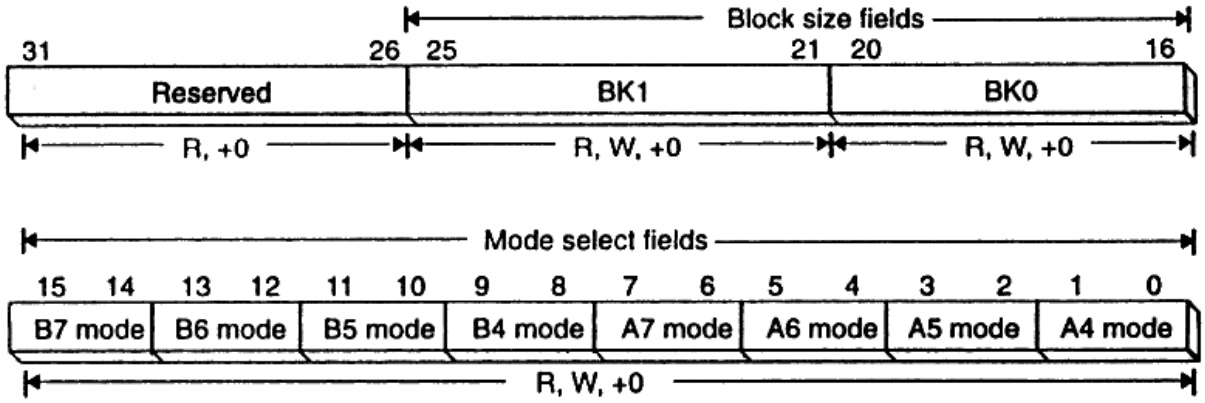
L'adressage circulaire est très utile dans plusieurs algorithmes DSP où les données en mémoire représentent une fenêtre glissante. Comme illustré sur la figure en haut de la page suivante, il permet la mise à jour des échantillons en écriture et en lecture sans utiliser le lourd mécanisme habituel de transfert des données entre emplacements mémoires.

Appliqué en temps réel au filtrage numérique où les données sont acquises en un flot continu d'échantillons, l'adressage circulaire procure un moyen très simple et élégant d'implémenter la ligne à retard.



Dans le C6713 tous les registres peuvent être utilisés pour l'adressage linéaire, cependant, seuls les huit registres **A4-A7** et **B4- B7** peuvent être utilisés comme pointeurs tournants, conjointement avec les deux unités .D,

Deux tampons circulaires indépendants sont disponibles, leur taille est donnée par les champs **BK0** et **BK1** du registre de contrôle **AMR Address Mode Register**.



Address mode register (AMR)

Description des champs Mode

Mode	Description
0 0	Adressage linéaire (défaut au reset)
0 1	Adressage circulaire, taille du tampon indiquée par BK0
1 0	Adressage circulaire, taille du tampon indiquée par BK1
1 1	Inutilisé

Le code ci-dessous illustre la préparation d'un tampon circulaire pointé par le registre **A5**, en utilisant le registre **B2** pour spécifier les valeurs appropriées dans **AMR**.

```
MVKL  .S2  0x0004,B2      ; lower 16 bits to B2. Select A5 as pointer
MVKH  .S2  0x0005,B2      ; upper 16 bits to B2. Select BK0, set N = 5
MVC   .S2  B2,AMR         ; move 32 bits of B2 to AMR
```

Les deux instructions **MVKL** et **MVKH** utilisant l'unité **.S**, écrivent successivement la valeur 0x0004 dans les 16 LSB du registre **B2** et 0x0005 dans les 16 MSB de **B2**. Une valeur 32 bits 0x0504 est donc contenue dans **B2**, elle est ensuite transférée dans le registre de contrôle **AMR** avec l'instruction **MVC**. L'instruction **MVC** (*Move Between Control File and Register File*) est la seule instruction qui peut accéder aux registres de contrôle, elle ne s'exécute qu'avec les unités fonctionnelles et les registres du côté B.

La valeur 0x0004 = 0100b dans les 16 LSB de **AMR** met la valeur 1 dans son $3^{ème}$ bit (b_2) et la valeur 0 dans tous les autres bits. Ceci met le mode du registre **A5** sur 01 et le sélectionne comme pointeur sur un tampon circulaire dont la taille est indiquée par le champ **BK0**.

La valeur 0x0005 = 0101b dans les 16 MSB de **AMR** correspond à une valeur N contenue dans le champ **BK0**. La taille en octets du tampon circulaire pointé par **A5** est égale à $\mathbf{2^{N+1}} = 2^6 = 64$ octets.

Les registres de contrôle du C6713 :

Registre de contrôle			addr	
AMR	Addressing mode register	Spécifie le mode d'adressage linéaire ou circulaire pour chacun des registres A4-A7 et B4-B7; Contient les tailles des tampons pour l'adressage circulaire	00000	R, W
CSR	Control status register	Contient le bit d'activation d'interruption global, les bits de contrôle de cache, et des bits d'état	00001	R W
IER	Interrupt enable register	Permet d'activer ou inhiber manuellement des interruptions individuelles	00110	R W
IFR	Interrupt flag register	Contient l'état des interruptions INT4-INT15 et NMI. Lorsque cette interruption se produit le bit correspondant est mis à 1, sinon il est à 0	00011	W
ISR	Interrupt set register	Permet d'activer manuellement les interruptions masquables INT15-INT4 dans le registre IFR. Écrire un 1 dans ISR met le flag correspondant dans IFR à 1. Écrire un 0 dans ISR n'a aucun effet. ISR n'affecte pas NMI et RESET.	00100	R W
ICR	Interrupt clear register	Permet d'inhiber manuellement les interruptions masquables INT15-INT4 dans le registre IFR.	00010	R
ISTP	Interrupt service table pointer	Pointe le début de la table de service d'interruption	00010	W
IRP	Interrupt return pointer	Contient l'adresse de retour d'une interruption masquable	00101	R W
NRP	NMI return pointer	Contient l'adresse de retour d'une interruption non masquable	00111	R W

Seule l'unité **.S2** peut lire et écrire dans les registres de contrôle par l'instruction **MVC**

Exercice didactique : Analyse d'un programme assembleur TMS320C6713 qui implémente le calcul de la fonction factoriel.

```
                    LDH     .D1   *A5,A4
                    MVK     .S1   A4,A1
                    SUB     .L1   A1,1,A1
LOOP :              MPY     .M1   A4,A1,A4
                    NOP
                    SUB     .L1   A1,1,A1
              [A1]  B       .S1   LOOP
                    NOP 5
                    STH     .D1   A4,*A6
```

A) Quelle est la fonction associée au registre **A1** dans ce programme

B) Justifier la raison de l'instruction **NOP** à la 5ème et à la 8ème ligne du programme

C) Décrire les opérations effectuées lors de l'exécution du programme

D) Trouver la valeur maximale que peut avoir x pour que le résultat de l'exécution de ce programme soit significatif

Solution

A) Le registre **A1** est défini comme un compteur de boucle. La boucle est la section du code commençant par le label **LOOP** et se terminant à l'instruction **B**,

B) L'instruction **MPY** a un slot de retard, et **B** cinq slots de retard. Le **NOP** qui suit oblige le traitement de ne se poursuivre que lorsque l'exécution de ces instructions est complètement achevée.

C) La valeur initiale de x est passée à **A4 (ligne 1)**, chargée dans **A1 (ligne 2)** et décrémentée **(ligne 3)** par l'instruction **SUB** à **(x – 1)**. La première instruction **MPY** réalise le produit $x(x-1)$ qui est accumulé dans le registre **A4 (ligne 4)**. A chaque itération dans la boucle, x est décrémentée **(ligne 6)**, multipliée par le contenu de **A4** et le produit est accumulée dans ce registre **(ligne 7)**, jusqu'à ce que **A1** = 0. La valeur contenue dans **A4** en sortie de la boucle est $x(x-1)\,(x-1)\cdot\cdot\cdot 2\cdot 1$ c'est le **factoriel x!** La valeur finale est stockée à l'adresse mémoire ***A6 (ligne 9)**

D) L'instruction **STH** sauvegarde un Half-Word signé, c'est-à-dire 16 bits dont un bit de signe, la valeur maximale est = 2^{15}=32768. La valeur maximale permise pour x est **7** car 7! =5040 alors que juste au dessus on a 8!=40320