Gestion des unités de mémorisation pour la synthèse d'architecture

Gwenolé Corre

N° d'ordre :

THESE

pour obtenir le grade de :

DOCTEUR DE L'UNIVERSITE DE BRETAGNE SUD

Spécialité **: Sciences de l'Ingénieur**

Mention **: Electronique et Informatique Industrielle**

Présentée et soutenue publiquement par

Gwenolé CORRE

Le 20 juin 2005

Gestion des unités de mémorisation pour la synthèse d'architecture

---

COMPOSITION DU JURY

Rapporteur	Ahmed Jerraya	TIMA, INPG de Grenoble
Rapporteur	Olivier Sentieys	ENSSAT, Université de Rennes 1
Examinateur	Elbey Bourenanne	LEII, Université de Bourgogne
Examinateur	Bruno Rouzeyre	LIRMM, Université de Montpellier II
Directeur de thèse	Eric Martin	LESTER, Université de Bretagne Sud
Examinateur	Eric Senn	LESTER, Université de Bretagne Sud

Laboratoire d'Electronique des Systèmes Temps Réel (LESTER)

Université de Bretagne Sud

Remerciements

Je remercie tout particulièrement Monsieur Eric Martin, Professeur des Universités de l'Université de Bretagne Sud et Directeur du laboratoire LESTER, pour m'avoir accueilli dans son équipe de recherche et accepté d'être mon directeur de thèse. Je le remercie également pour les conseils qu'il m'a apportés tout au long de ces trois années.

Je tiens à remercier Madame Nathalie Julien-Talon et Monsieur Eric Senn, Maîtres de Conférences à l'Université de Bretagne Sud pour leur disponibilité et leur encadrement. Je remercie également Messieurs Pierre Bomel et Dominique Heller, Ingénieurs de Recherche au laboratoire LESTER, pour leurs précieux conseils et leur participation au développement de l'unité de mémorisation de l'outil de synthèse d'architecture GAUT.

Je remercie Monsieur Ahmed Amine Jerraya, Directeur de recherche CNRS au laboratoire TIMA de l'INPG de Grenoble et Monsieur Olivier Sentieys, Professeur des Universités à l'ENSSAT, Université de Rennes I, pour avoir accepté d'être rapporteurs de ce travail et membres du jury.

Je tiens également à remercier pour leur participation à ce jury Monsieur Elbey Bourennane, Professeur des Universités au laboratoire LEII, Université de Bourgogne et Monsieur Bruno Rouzeyre, Professeur des Universités au laboratoire LIRMM, université de Montpellier II.

Enfin, je remercie tous les membres du LESTER.

A Olwen et Anaëlle

Résumé

Les techniques et les capacités d'intégration ne cessent de croître ; les systèmes peuvent traiter des applications de plus en plus complexes. Si les capacités de calcul augmentent fortement, il n'en va pas de même pour les capacités de stockage qui forment un goulet d'étranglement à mesure que la quantité d'information à traiter augmente. Au vu de cette augmentation, il est important de bien gérer les unités de mémorisation lors de la conception de systèmes. Le problème de gestion des données et de la mémoire peut être traité au cours des différentes étapes du flot de conception. Si des optimisations sont possibles à toutes les étapes, les opportunités d'optimisation sont plus importantes à haut niveau.

Dans cette thèse, nous proposons une méthodologie de gestion des unités de mémorisation intégrée au flot de synthèse d'architecture. Cette méthodologie et le formalisme qui y est associé permettent d'exploiter les opportunités d'optimisation des unités de mémorisation disponibles à ce niveau d'abstraction. Cette méthodologie s'intègre dans une stratégie globale de gestion de la mémorisation actuellement développée au laboratoire LESTER. Elle répond aux besoins fonctionnels et aux contraintes applicatives et s'articule autour de la synthèse d'architecture et de l'outil *GAUT*. La stratégie se décompose en trois phases. La première, en amont de la synthèse d'architecture, explore la hiérarchie mémoire et la distribution de données structurées des applications. Elle permet de réaliser efficacement et rapidement une optimisation à gros grain pour les applications en traitement du signal et de l'image. Cette distribution est exploitée lors de l'étape de synthèse haut niveau sous contrainte de mémorisation, objet de ce travail de thèse. Enfin, la phase en aval de la synthèse de haut niveau consiste à concevoir les générateurs d'adresses de l'unité de mémorisation.

A partir des contraintes, de nouvelles approches ont été développées pour améliorer la gestion de la mémorisation : d'une part, une analyse des données manipulées dans les applications de traitement du signal a permis de mettre en œuvre un nouveau mécanisme de placement des données et de génération d'adresses ; d'autre part, nous avons introduit une nouvelle gestion des accès mémoire basée sur le modèle de gestion de production kanban. Cette nouvelle gestion permet de répondre au problème lié à la violation des contraintes temporelles et fonctionnelles qui peut être engendrée par l'introduction de contraintes mémoires.

Ces développements reposent sur une approche formelle, pour la spécification de contraintes d'accès aux données, l'optimisation temporelle de leur accès et la génération de l'unité de mémorisation. Ainsi, un ensemble de graphes permet la représentation des propriétés particulières à chaque étape de la synthèse.

Les résultats de synthèses illustrent et valident les choix définis dans notre approche de gestion de la mémorisation. Un premier champ d'expériences montre les possibilités de synthèse offertes par l'outil et l'apport de la synthèse sous contrainte de mémorisation. Le second champ d'expérimentation illustre l'apport du mécanisme de gestion du vieillissement des données. Il relève également l'intérêt de l'introduction du mode de gestion kanban des accès mémoire pour une application manipulant un nombre de données important ou ayant des contraintes temporelles sévères. Enfin, la dernière expérience illustre la capacité de notre approche et de notre outil à traiter des problèmes industriels complexes.

Le travail réalisé au cours de cette thèse a permis, dans un premier temps, de définir une méthodologie de synthèse sous contrainte de mémorisation, de l'intégrer et de la valider dans l'outil de synthèse d'architecture *GAUT*. Cet outil est diffusé et exploité par de nombreux partenaires universitaires et industriels. Dans un second temps nous avons développé des modèles de gestion des données en tenant compte des spécificités du domaine d'application en traitement du signal et de l'image. Ces travaux ont été intégrés au flot de synthèse d'architectures et à l'outil *GAUT*, mais notre démarche de gestion des accès mémoire par kanban pourrait être étendue à un spectre plus large que le domaine de la synthèse d'architecture.

Abstract

Techniques and integration capacities increase. Systems handle more and more complex applications. Processing increases faster than storage capacities. Memory becomes a bottleneck since the quantity of information increases. In this context, it is crucial to efficiently manage memory all along the design flow. Good optimization opportunities are offered at the high level synthesis step.

We propose a methodology to integrate the management of memory unit into our high level synthesis flow. The methodology and its formalism allow to exploit optimization opportunities. The methodology is integrated in a global memory unit design flow developed in our laboratory. First, we extract the memory hierarchy and data distribution from the data structure from an input specification; this allows to carry out a coarse grain optimization on the data distribution. This distribution will be integrated as a new constraint in our high level synthesis design flow. Finally, we design the address generators of the memory unit.

From the memory constraints, a first approach of synthesis is proposed. This approach was integrated into our HLS tool, *GAUT*. New approaches were developed to improve memory management. An analysis of the data handled in digital and signal processing applications leads to a generic memory architecture to store specific data. Then, we introduced a new memory access management based on the kanban system. This new management gives a solution to the violation of temporal constraints involved by the introduction of memory constraints. The synthesis results illustrate and validate the choices defined in our memory management approach. The first experiment field shows the synthesis possibilities offered by our tool and the contribution of synthesis under memory constraints. The second experiment field shows the impact of management of the ageing data and memory accesses managed by kanban for an application handling a significant number of data. The last experiment illustrates the capacity of our approach and our tool to deal with complex industrial problems.

We define a methodology of synthesis under memory constraints and validate it with our HLS tool *GAUT*. Then, we developed models, and implementation techniques, for specific data management in digital and signal processing. This work was also integrated into our high level design flow and tool and the proposed methodology could be extended to others domains.

Table des matières

Table des Figures

Table des Tableaux

Glossaire

ACG	Algorithmic Constraint Graph
ALAP	As Last As Possible
ASAP	As Soon As Possible
ASIC	Application Specific Integrated Circuit
ATOMIUM	**A T**oolbox for **O**ptimising **M**emory **I**/O **U**sing geometrical **M**odels
BLMS	Bloc Least mean Square
CDFG	Control and Data Flow Graph
CISC	Complete Instruction Set Computer
DFG	Data Flow Graph
DRAM	Dynamic Random Access memory
DSP	Digital Signal Processor
DTSE	Data Transfer and Storage Exploration
DWT	Discret Wavelet Transform
FFT	Fast Fourier Transform
FIFO	First In First Out
FIR	Finite Impulse Reponse
FPGA	Field Programmable Gate Array
FRAM	Ferroelectric Random Access memory
FSM	Finite State Machine
GAUM	Génération Automatique d'Unité de Mémorisation
GAUT	Génération Automatique d'Unité de Traitement
GCA	Graphe de Conflit d'Accès rotatif
GCG	Global Conflict Graph
GFS	Graphe Flot de Signaux
GSAL	Graphe de Séquence d'Adressage Logique
GSU	Graphe de Séquences Unifiées
GVA	Graphe de Vieillissement des Adresses logique
ILP	Integer Linear Programing
IOCG	Input output Constraint Graph
IP	Intellectual Properties
LFU	Less Frequently used

LIFO	Last In First Out
LMS	Least Mean Square
LRU	Last Recently Used
MCG	Memory Constraint Graph
MIMD	Multiple Instruction Multiple Data
MISD	Multiple Instruction Single Data
MRAM	Magnetic Random Access memory
NLMS	Normalized Least Mean Square
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
RMW	Read-Modify-Write
ROM	Read Only Memory
RTL	Register Transfer Level
SFG	Signal Flow Graph
SIMD	Single Instruction Multiple Data
SISD	Single Instruction Single Data
SoC	System on Chip
SRAM	Static Random Access memory
TDSI	Traitement Du Signal et de l'Image
UAL	Unité Arithmétique et Logique
UCOM	Unité de COMmunication
UM	Unité de mémorisation
UT	Unité de Traitement
VHDL	VHSIC (Very High Speed Integrated Circuit) Hardware Description Language

Chapitre I
Introduction

Les techniques et les capacités d'intégration ne cessent de croître au cours des années et permettent d'intégrer des applications de plus en plus complexes. Si les capacités de calcul augmentent fortement, il n'en va pas de même pour les capacités de stockage des données. Les unités de mémorisation forment un goulet d'étranglement qui devient de plus en plus critique à mesure que la quantité d'information augmente.

Dans cette introduction, nous définirons d'abord les unités de gestion mémoire (UGM). Nous introduirons ensuite le contexte des applications orientées vers le TDSI, en exhibant leurs propriétés spécifiques et leurs influences sur la mémoire. Nous nous positionnerons ensuite dans le flot de conception et détaillerons les principales étapes de la synthèse comportementale. Enfin nous introduirons le plan de ce mémoire.

Le marché de l'électronique s'oriente pour une grande part vers des produits grand public visant principalement des applications multimédia ou télécommunications (PDA, jeux vidéo, téléphones mobiles, GPS, …). La croissance des besoins est soutenue par l'augmentation des capacités d'intégration : *l'International Technology Roadmap for Semiconductors* [ITRS03] prévoit que le nombre de transistors atteindra les 97 millions en 2007 et 1,5 milliards en 2013. L'évolution des technologies devrait permettre, à terme, l'intégration de toutes les fonctions du domaine du multimédia et des télécommunications dans un seul appareil portable.

Les applications et fonctions seront donc intégrées dans une seule et unique puce (System on Chip) contenant des parties logicielles et matérielles. Compte tenu des contraintes de consommation électrique et des performances temporelles des systèmes embarqués futurs, la conception de circuits numériques devra intégrer des accélérateurs matériels pour le traitement intensif. Par ailleurs, comme les fonctions multimédia et de télécommunication manipulent des quantités d'information et de données importantes, les besoins de stockage et de transferts de données seront donc de plus en plus critiques. Or, les progrès technologiques des mémoires ne permettent pas de combler l'écart en terme de performance, de consommation et de surface avec les unités de traitement.

Pour développer les futurs systèmes, les équipes de recherche proposent donc des méthodologies de conception qui intègrent la gestion de la mémorisation. Ces méthodologies tendent à réaliser des compromis entre différents aspects (calcul/mémorisation, calcul/communication, besoins/contraintes/choix d'implantation). Elles permettent de réduire le fossé entre les performances des unités de traitement et celles des mémoires et s'appuient sur une gestion efficace de la mémoire.

I.1. Les unités de gestion mémoire

La quantité de données à transférer et à stocker pour le traitement des applications temps réel exige des performances qui ne peuvent être atteintes par les débits d'accès aux mémoires actuelles. Pour pallier ce problème, il faut :

- concevoir des architectures d'unité de mémorisation efficaces ; les unités de mémorisation intègrent les ressources matérielles nécessaires à mémoriser et à transférer les données utiles pour l'exécution d'un programme ou d'une application.
- adapter la gestion des mémoires ; une gestion efficace de la mémoire intègre une distribution et un placement des données structurées et des mécanismes d'accès permettant d'estomper ses limitations.

Il existe de nombreuses techniques et méthodologies, à tous les niveaux d'abstraction du flot de conception, pour améliorer les performances des mémoires. Les unités de gestion mémoire (UGM) permettent, pour des architectures mémoire définies, de combler l'écart de performance

avec les unités de traitement. Les principales solutions intégrées aux UGM comprennent des dispositifs électroniques et/ou logiciels qui permettent de stocker et transférer les données manipulées par les unités de traitement. La gestion de la mémoire repose sur deux modèles architecturaux, le modèle Von Neumann et le modèle Harvard.

- Le modèle Von Neumann : ce modèle est caractérisé par un processeur et une mémoire reliés par un bus d'adresse et un bus de données. Les instructions et les données sont stockées dans la mémoire. La mémoire est ainsi très sollicitée ; elle représente le principal frein aux performances du processeur. Un certain nombre de techniques telle que la hiérarchie peuvent être développées pour remédier au problème créé par l'écart entre la vitesse des CPU et le temps d'accès à la mémoire.

- Le modèle Harvard : il est caractérisé par un processeur et deux mémoires distinctes, une mémoire de programme et une mémoire de données. Chaque mémoire est reliée au processeur par un bus de données et un bus d'adresses. Ce modèle accélère la vitesse du processeur en parallélisant d'accès mémoire parallèles entraînant une. Les architectures mémoire pour les DSP à hautes performances intègrent généralement une mémoire de programme et deux mémoires de données ce qui permet d'obtenir un gain de performance en terme de nombre d'accès mais entraîne un surcoût matériel.

Les systèmes d'exploitation intègrent également des mécanismes de gestion de la mémoire. Ces mécanismes reposent sur le contrôle et l'assignation de ressources mémoire disponibles pour les différentes tâches s'exécutant.

I.2. Spécificité du Traitement Du Signal et de l'Image

Dans cette étude, nous nous intéressons aux applications de traitement du signal et de l'image (TDSI): filtrages (FIR,LMS,…), transformations (DCT,FFT,…), codages d'erreurs (Turbo code, viterbi,…), compression et décompression d'image et de flux vidéo (JPEG, MPEG)….

Ces applications présentent certaines caractéristiques communes que nous exploitons dans notre approche de gestion des unités de mémorisation intégrée à la synthèse de haut niveau :

- **Déterminisme :** la plupart de ces applications sont dites déterministes au sens de leur exécution, c'est à dire que l'on peut déterminer a priori les séquences de calcul et les transferts de données. En général, une grande partie des applications TDSI ont un comportement dépendant de paramètres connus a priori.

- **Fort volume de données :** la fonctionnalité des systèmes est basée sur le traitement intensif de flux de données. Le concepteur du système devra donc se concentrer sur l'utilisation d'architectures de communication évitant la congestion des transferts sur les

média de communication et dans les tampons tout en réduisant le coût du stockage des données.

- **Flux de données multidimensionnelles :** le traitement de l'information est typiquement réalisé sur des flux de données structurées représentés sous la forme de matrices mono ou multidimensionnelles.
- **Motifs de calcul répétitifs :** les traitements réalisés dans les applications TDSI ont une structure régulière. La description des opérations réalisées sur des tableaux multidimensionnels est spécifiée à l'aide de boucles imbriquées permettant la répétition d'un motif de calcul.
- **Temps réel :** la nature temps réel des applications du domaine ajoute des contraintes fortes sur les phases de spécification, d'exploration architecturale et d'intégration visant les systèmes embarqués.
- **Contraintes de conception :** les applications sont fortement utilisées dans des produits grand public qui requièrent un délai de mise sur le marché extrêmement court. A cela s'ajoutent des contraintes drastiques de consommation et de surface lorsque les systèmes sont embarqués dans des appareils électroniques portatifs.

Nous introduirons dans le chapitre suivant les techniques et méthodologies de gestion des unités de mémorisation inhérentes aux propriétés du domaine du traitement du signal.

I.2.1. Données structurées et typées

Dans le domaine particulier du traitement du signal, on peut extraire un certain nombre de généralités sur les données manipulées. Les données sont dites structurées, car elles sont régulières au niveau des objets globaux (souvent des tableaux multi-dimensionnels éventuellement infinis) et au niveau des motifs de calcul (partitionnement régulier des tableaux). Pour des traitements d'images bas niveaux, les calculs consistent souvent à appliquer des traitements locaux relativement indépendants sur toutes les parties de tableaux.

Les données structurées sont donc très bien adaptées au parallélisme de données ; les structures sont simples et les calculs répétitifs. La complexité n'est pas dans l'expression de l'algorithme, car c'est souvent une expression algébrique relativement simple itérée sur tout ou partie des données structurées.

I.2.2. Parallélisme de traitement et d'accès aux données

Dans le cas des applications TDSI, on peut retrouver différentes classes de parallélisme.

- Le **parallélisme de tâches** : on assimile le plus souvent une tâche à un algorithme. C'est surtout aux niveaux des chaînes de traitement qu'apparaît le parallélisme de tâches, par exemple des chaînes de traitement du signal et des chaînes de communication numérique. Les accès se font à la marge des tâches et souvent à des données structurées différentes.
- Le **parallélisme de données** : les unités de calcul effectuent toutes la même action mais sur un ensemble de données différentes. En traitement d'image, des traitements locaux sont appliqués et des éléments d'une même structure de données peuvent être accédés en parallèle. Dans ce cas, la distribution des éléments d'une même structure de données aura une influence sur le parallélisme d'accès. Les données placées dans la même mémoire ne pourront être accédées en parallèle que si l'architecture de la mémoire le permet (mémoire double port par exemple).
- Le **parallélisme d'opérations**: en grain fin, plusieurs traitements locaux à un même algorithme peuvent être effectués. C'est le cas typique de la multiplication-accumulation du produit de convolution. Ces traitements sont réalisés pour des architectures de type VLIW, super pipeline ou super scalaire. Toutes ces architectures permettent d'effectuer des opérations en parallèle, pour cela il faut que leurs opérandes soient accessibles. La distribution des éléments de structures de données en mémoire est donc très importante et doit être gérée de manière efficace pour permettre d'exploiter au mieux les opportunités de parallélisme d'opérations qui sont offertes par ces architectures.

Le volume de données à gérer en mémoire est, pour ces applications, très important. Pour exploiter au mieux ces différentes classes de parallélisme, il est nécessaire de les intégrer dans le flot de conception.

I.3. Méthodologies de conception

La complexité des applications nécessite la mise en place de démarches de co-conception logicielle/matérielle (Hardware/Software codesign). Les applications sont décomposées en un ensemble de fonctions. Une étape de partitionnement permet de répartir les fonctions vers le matériel et/ou le logiciel.

Les solutions logicielles s'appuient, dans le domaine du traitement du signal et de l'image, sur des architectures générales ou spécifiques comme les processeurs de traitement du signal (DSP). L'exploitation de solutions logicielles repose sur l'adaptation d'un algorithme à une architecture cible. Pour obtenir une implantation optimale, il est nécessaire d'analyser l'algorithme pour en extraire le parallélisme et utiliser au mieux les capacités de traitement de la cible [Mart96].

Les solutions matérielles créent des architectures dédiées aux applications. Cette démarche s'inscrit dans une conception descendante qui consiste à raffiner la description d'une application pour atteindre une architecture adaptée. A partir d'une spécification algorithmique et d'un ensemble de contraintes, on effectue différentes étapes de synthèse visant une solution ASIC ou FPGA. L'objectif est de minimiser les coûts (surface, consommation) et de répondre aux contraintes temporelles fixées.

Dans le cadre de cette thèse, nous visons des solutions matérielles pour des applications en TDSI. Notre travail se situe au niveau de la synthèse d'architecture (ou synthèse de haut niveau) qui s'inscrit dans le flot de conception système présenté par Gajski [Gajs91]. Ce modèle en Y permet une représentation complète de tous les niveaux et domaines de la spécification système et les différentes opérations de transformations possibles. La synthèse de haut niveau permet la transformation d'une description de niveau algorithmique en une description de niveau RTL. Les architectures synthétisées sont couramment décomposées en quatre unités fonctionnelles. L'**unité de traitement** réalise l'ensemble des calculs de la spécification algorithmique. L'**unité de mémorisation** permet de stocker les données nécessaires aux traitements. L'**unité de communication** gère les transferts avec l'extérieur. L'**unité de contrôle**, hiérarchique ou locale, pilote les unités fonctionnelles.

La conception de chaque unité fonctionnelle propage des contraintes vers les autres unités. Une approche globale permettrait d'optimiser les architectures. Cependant, les contraintes fonctionnelles (interdépendances) rendent une approche globale trop complexe. Il faut donc définir un ordre de conception. Or, l'ordre de synthèse entre unité de traitement et unité de mémorisation est très important lors de la synthèse de spécification algorithmique d'applications TDSI ; le volume de données à mémoriser et à traiter est important et il faut respecter les contraintes temps réel. De ce fait, la synthèse du traitement peut conduire à imposer des contraintes pénalisantes pour l'unité de mémorisation.

I.4. Orientation

Les possibilités de mémorisation sont nombreuses. Pour gérer au mieux la grande quantité de données des applications de TDSI, il est nécessaire de mettre en œuvre une gestion des unités de mémorisation en tenant compte des spécificités de ce domaine d'application.

Le travail de cette thèse développe la gestion et l'optimisation des unités de mémorisation pour la synthèse de haut niveau. Nous proposons une méthodologie de gestion automatique et reproductible des unités de mémorisation afin d'obtenir des solutions architecturales efficaces pour des applications de traitement du signal et de l'image. Cette méthodologie repose sur une approche formelle à base de graphe ; pour la représentation et l'exploitation systématique des propriétés d'accès aux données à mémoriser.

I.5. Plan du mémoire

Le chapitre II présente un état de l'art sur les techniques permettant de gérer de larges quantités de données. Nous nous intéresserons plus particulièrement aux solutions apportées au niveau architectural. Puis, nous présenterons les solutions développées autour de la synthèse de haut niveau.

Le chapitre III présente la stratégie globale de conception de l'unité mémoire développée au laboratoire et son intégration dans la synthèse de haut niveau. Les propriétés des données qui devront être gérées par l'unité de mémorisation seront détaillées.

Le chapitre IV définit l'approche formelle et les modèles à base de graphes pour la synthèse sous contraintes de mémorisation. Nous modéliserons les contraintes de mémorisation et nous introduirons différents modèles de gestion des accès mémoire, et les transformations à opérer pour la synthèse.

Le chapitre V présente une optimisation originale de l'ordonnancement, améliorée en flux tiré par la méthode kanban.

Le chapitre VI présente l'application de nos approches pour des applications TDSI et mettra en avant l'intérêt des modèles de gestion mémoire proposés. Nous illustrerons la capacité de nos approches et de notre outil à traiter des problèmes de grande complexité et manipulant de grande quantité de données.

Le dernier chapitre conclura ce manuscrit résumant les apports de notre approche et les perspectives qui s'en dégagent.

Chapitre II
Etat de l'art

L'accès rapide à de grands volumes de données forme dans les systèmes électroniques un goulet d'étranglement qui limite les performances en termes de quantité de calcul. Le problème de l'optimisation de la mémoire est présent dans toutes les étapes du flot de conception systèmes. Des solutions peuvent être apportées aux différents niveaux de conception et dans l'exploitation des systèmes. Le concepteur fixera ainsi des compromis entre surface, rapidité, volume et consommation et ceci à tous les niveaux de conception : application, architecture, algorithme, niveau logique ou technologique. Dans cet état de l'art nous nous intéresserons spécifiquement aux approches architecturales, pour la définition de structures architecturales et de fonctionnements optimisés, que les concepteurs intègrent dans la conception des circuits et systèmes électroniques. Nous compléterons cet état de l'art par une présentation des méthodologies de gestion de la mémorisation intégrée aux flots et aux outils de conception haut niveau.

II.1. Accès rapides à la mémoire pour les traitements

II.1.1. Accroissement du fossé entre mémoire et traitement, quelques domaines d'optimisation.

Les unités de mémorisation deviennent de plus en plus un goulet d'étranglement à mesure que la quantité d'information à stocker s'accroît. Les performances temporelles entre processeurs et mémoire sont équivalentes au début des années 80 (temps de cycle de 120 ns pour les processeurs contre temps d'accès de 140 ns pour les mémoires DRAM), l'écart ne cesse pas de croître. Ce phénomène est illustré par la Figure II.1 qui montre l'écart se creusant entre les performances des unités de traitement et celles des mémoires.

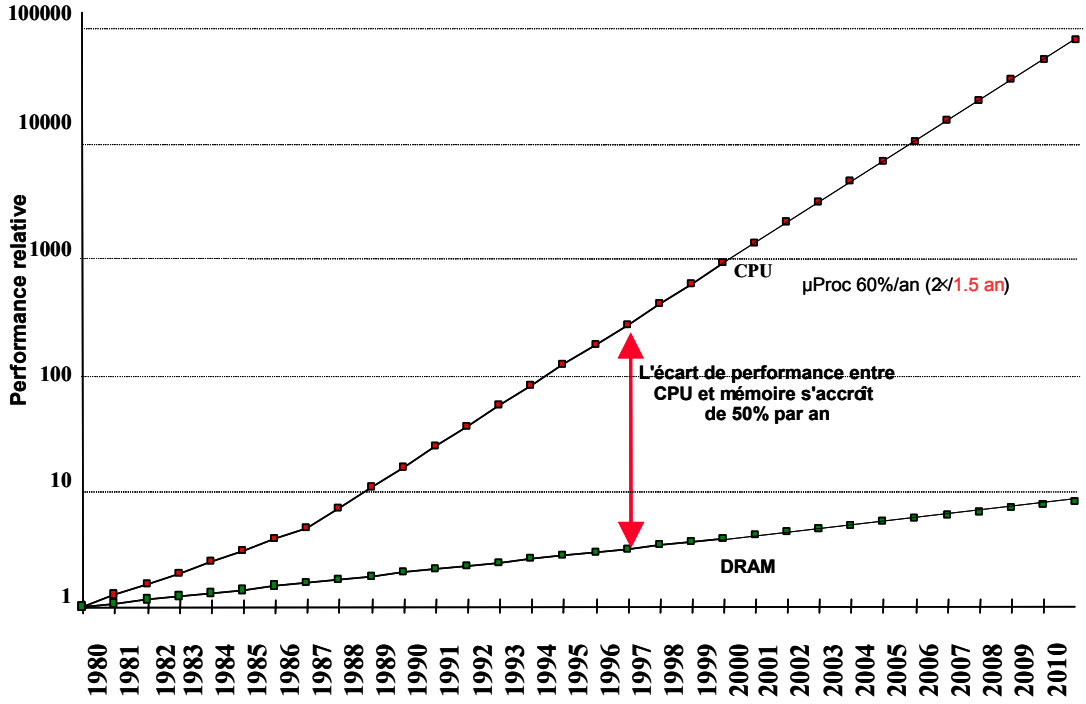


Figure II.1 : Evolution de l'écart de performances (latence) entre CPU et mémoire

Le goulet d'étranglement que représente la mémoire ne se limite pas aux performances. La prédominance des mémoires sur la surface et la consommation globale des systèmes va s'accentuer avec l'augmentation des quantités d'information des nouvelles applications. Ainsi, l'ITRS [ITRS03] prévoit que la surface de la mémoire sur des systèmes sur puce (SoC, System On Chip) atteindra plus de 90% dans une dizaine d'années (Figure II.2). La consommation en puissance des mémoires représente actuellement 10 à 15% de la consommation pour les architectures des processeurs et peut atteindre près de 50% de la consommation globale pour des applications de télécommunication ou multimédia [Eric01].

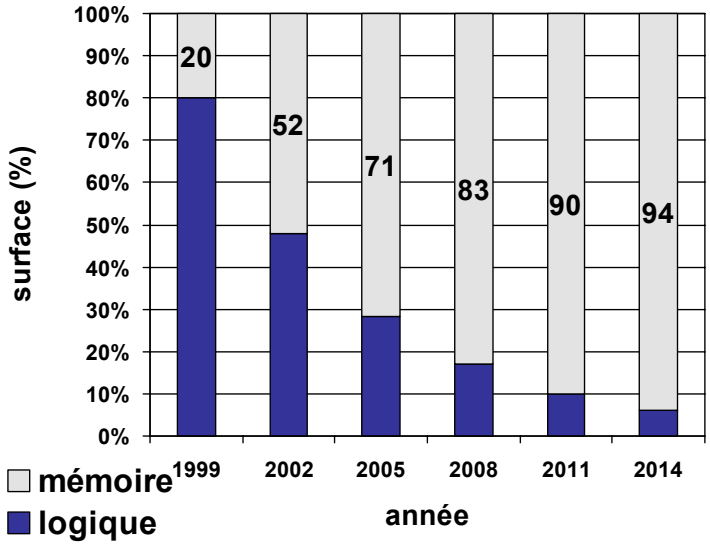


Figure II.2 : Evolution de la surface des SoC

Pour répondre aux différents problèmes de performance, surface et consommation inhérents aux mémoires, les concepteurs peuvent agir tout au long du flot de conception pour optimiser les unités de mémorisation. Nous positionnerons les solutions pour la conception mémoire aux différents niveaux d'abstraction du flot de conception. Le niveau architectural qui est au cœur de ce travail de thèse sera détaillé dans la section II.3.

a. Le niveau technologique

L'objectif des travaux menés au niveau technologique est de réaliser des mémoires aussi rapides que les SRAM et aussi économiques (capacité vs coût) que les disques durs. Si pour l'instant les mémoires SRAM, DRAM et FLASH représentent plus de 90% des ventes de mémoires sur le marché mondial, l'émergence de nouvelles technologies pourrait bien bouleverser l'ordre établi.

Les mémoires SRAM (Static Random Access Memory): les mémoires SRAM sont des mémoires dites volatiles. Elles sont caractérisées par leurs vitesses d'accès aux données. Un point de mémorisation est composé de 6 transistors ce qui offre une densité d'intégration faible.

Les mémoires DRAM (Dynamic Random Access Memory): ces mémoires offrent une bonne densité (point de mémorisation composée d'un transistor et d'une capacité) mais sont plus lentes que les mémoires SRAM. De plus, les données doivent être rafraîchies.

Les mémoires FLASH : les mémoires FLASH sont composées d'un unique transistor ; les écritures sont très lentes par rapport aux mémoire DRAM et SRAM.

De nouvelles technologies apparaissent comme les mémoires FRAM (Ferroelectric Random Access Memory) et MRAM (Magnetic Random Access Memory). Les mémoires FRAM utilisent un film ferro-électrique comme diélectrique de capacité servant à stocker une donnée. Le film ferro-électrique est polarisé suivant le sens du champ électrique appliqué. La fonction de stockage tire avantage de cette caractéristique qui assure la non-volatilité ainsi que des temps d'accès relativement courts. Les mémoires MRAM utilisent une charge magnétique, Les données sont stockées entre deux couches ferro-magnétiques. Elles allient les avantages de la SRAM (vitesse), de la DRAM (haute capacité, coût réduit) et de la mémoire Flash (non volatilité), tout en consommant très peu. *Phase change*, les *Floating body* DRAM, les mémoires *Single/Few Electron* et les mémoires *moléculaires* sont à l'étude et commenceront à apparaîtrent d'ici cinq à dix ans [ITRS03].

b. Le niveau architectural

Les optimisations pouvant être effectuées à ce niveau seront détaillées par la suite dans la section II.3.

c. Le niveau système

Les architectures sont classées suivant la classification de [Flyn72]. Elles se décomposent en quatre groupes qui dépendent du flot d'instruction et du flot de données. Le Tableau II-1 répertorie les différentes classes d'architecture de systèmes.

		Flot de données	
		Simple	Multiple
Flot d'instructions	Simple	**SISD**	**SIMD**
	Multiple	**MISD**	**MIMD**

Tableau II-1 : Classification de Flynn

SISD (Single Instruction Single Data) : un seul flot d'instructions, un seul flot de données. Cette catégorie correspond aux architectures monoprocesseur.

SIMD (Single Instruction Multiple Data): un seul flot d'instructions, plusieurs flots de données. Une instruction est exécutée par plusieurs processeurs qui utilisent différents flots de données. Chaque processeur possède sa propre mémoire de données, et un seul processeur réalise le contrôle des instructions à partir d'une mémoire d'instruction unique. Les architecture vectorielles constituent la majeure partie des processeurs SIMD.

MISD (Multiple Instruction Single Data): plusieurs flots d'instructions, un seul flot de données. Aucune architecture multi-processeur n'est construite sur ce type.

MIMD (Multiple Instruction Multiple Data): plusieurs flots d'instructions, plusieurs flots de données. Chaque processeur manipule ses propres instructions et utilise ses propres données. Les architectures de système multi-processeurs utilisent souvent des processeurs standards.

Les architectures mémoire des systèmes utilisant des architectures MIMD s'organisent autour de mémoires partagées et de mémoires distribuées.

Les systèmes à mémoire partagée : les mémoires partagées sont utilisées dans des systèmes comportant un nombre d'unités de traitement relativement faible. L'unité de mémorisation est généralement organisée de manière hiérarchique avec des mémoires caches de taille importante. Les temps d'accès à la mémoire sont uniformes puisque les systèmes ne possèdent qu'une seule mémoire principale. Ils ont pour avantage une gestion simple du partage des données entre les différentes ressources à travers un mécanisme uniforme de lecture et d'écriture des structures de données dans la mémoire principale.

Les systèmes à mémoire distribuée : chaque unité de traitement possède sa propre mémoire principale. Les nœuds (unité de traitement + mémoire) sont reliés en utilisant des réseaux d'interconnexion. Les temps d'accès aux mémoires ne sont pas uniformes. En optant pour ce système, l'utilisateur doit apporter une attention particulière à la distribution des données ainsi qu'à la gestion des communications. Le Tableau II-2 répertorie les points différenciant les deux systèmes.

	Système à mémoire partagée	Système à mémoire distribuée
Temps d'accès	Uniformes	Non uniformes
Nombre de processeurs	Limité	Grand
Communication	Simple	Complexe
Flexibilité	Peu flexible	Flexible
Type d'interconnexion	Bus	Réseau d'interconnexions
Organisation de la mémoire	Plusieurs niveaux de cache	Cache simple
Taille de la mémoire	Mémoires de grandes tailles physiquement centralisées	Mémoires de petites tailles physiquement centralisées

Tableau II-2 : Comparaison mémoire partagée / mémoire distribuée

L'écart se creusant entre les unités de traitements et les unités de mémorisation nécessite la mise en œuvre de techniques de gestion pour réaliser des accès rapides à la mémoire. Les accès devront être mieux gérés entre les unités de mémorisation et les unités de traitement mais aussi entre les différentes mémoires des unités de mémorisation. Une solution pour combler l'écart entre les processeurs et les mémoires est l'introduction de la hiérarchie mémoire.

II.1.2. La hiérarchie mémoire

a. Généralités :

Une solution pour palier la lenteur de la mémoire, comparée aux vitesses grandissantes des unités de traitement, a été proposée très tôt par les concepteurs de processeurs. La mémoire a été hiérarchisée en différents niveaux de tailles et de temps d'accès décroissant au fur et à mesure que l'on se rapproche de l'unité de traitement (Figure II.3). Les hiérarchies mémoire peuvent être composées de plusieurs niveaux hiérarchiques et distribuées sur plusieurs bus. Le premier niveau en dehors des files de registres intégrées à l'unité de traitement est constitué de mémoires caches. De nombreux ouvrages traitent de la conception de hiérarchie mémoire [Henn02], [Kulk01].

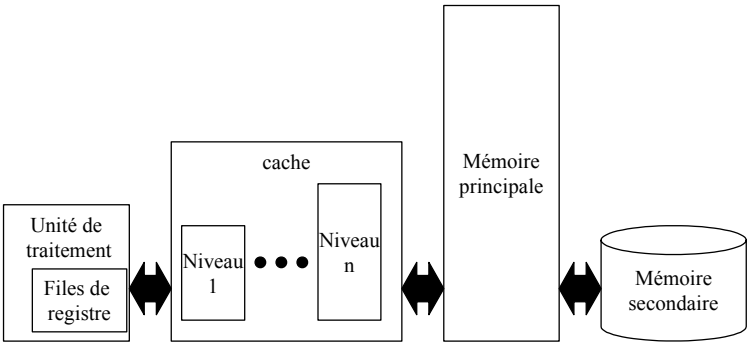


Figure II.3 : Hiérarchie mémoire

Dans une hiérarchie mémoire classique, les données ne peuvent être recopiées d'un niveau à un autre que s'ils sont adjacents. La Figure II.3 représente une architecture simple avec n niveaux de hiérarchie mémoire. Elle est composée d'une mémoire cache et d'une mémoire principale. La mémoire cache s'interface entre l'unité de traitement et la mémoire principale. Plus le niveau hiérarchique diminue (plus on se rapproche de l'unité de traitement), plus les mémoires mises en œuvre sont chères et petites, en contre partie, elles sont plus rapides. L'unité de traitement sollicite le cache et non directement la mémoire principale. Si une copie des données est disponible dans le cache il peut répondre immédiatement. Dans le cas contraire, le cache doit accéder à un niveau inférieur de la hiérarchie, copier le bloc contenant la donnée et ensuite répondre à l'unité de traitement.

Le principe des caches repose sur la localité spatiale et temporelle des données accédées dans une application. La plupart des applications contiennent des boucles, spécialement les applications en traitement du signal et de l'image. Ceci implique des accès répétés aux données et met en avant une localité temporelle forte. De la même manière, les données accédées séquentiellement font apparaître la localité spatiale pouvant être exploitée dans la gestion des accès. Les variables présentent une grande variété de motifs d'accès et de types de localités suivant les applications spécifiées et le type de données utilisées. Les scalaires présentent une

grande localité temporelle et une localité spatiale modérée. Les vecteurs présentent une localité spatiale forte, et peuvent avoir ou pas de localité temporelle. Quelques recherches se proposent de décomposer le cache entre un cache spatial et un cache temporel qui stocke les données structurées ayant respectivement une localité spatiale ou temporelle forte. Cette approche revient à un mécanisme de prédiction dynamique pour diriger les données soit dans le cache temporel, soit dans le cache spatial, en se basant sur l'historique des durées de vie des données en registre. Dans le contexte des systèmes embarqués, l'approche de [Grun01] utilise une architecture similaire de division du cache. Elle alloue les variables statiquement à différents modules mémoire locaux, éliminant ainsi le surcoût en surface et consommation des mécanismes de prédiction dynamiques. Donc en ciblant les types spécifiques de localités des différentes variables, une meilleure utilisation de la bande passante mémoire peut être mise en œuvre. Les extractions inutiles dues à la non-adaptation des données sont éliminées. Par exemple, si une variable, avec une faible localité spatiale, est accédée par un cache avec une taille de bloc important, un grand nombre de valeurs lues en mémoire principales ne sont jamais utilisées. Des politiques de gestion de remplacement des blocs de données sont mises en œuvre pour exploiter les localités des données. Les performances de la hiérarchie mémoire dépendent de l'efficacité de ces politiques.

L'objectif de la conception d'une hiérarchie mémoire est double. D'une part, il s'agit de réduire les temps d'accès à l'unité de mémorisation pour diminuer la latence des applications ; d'autre part, il s'agit de réduire la puissance dissipée par accès notamment pour les applications embarquées.

b. La mémoire cache

Un cache est le plus souvent intégré au processeur et possède des temps d'accès relativement rapides. Un cache est décomposé en blocs qui contiennent un ensemble de données fixes

On distingue trois types de gestion des accès en mémoire cache. Ils se définissent suivant leur degré d'associativité.

- Le cache à adressage direct (*Direct Mapped Cache*) : un bloc est recopié à une adresse unique en fonction de sa provenance. Cette adresse est définie par une opération de modulo sur l'adresse du niveau hiérarchique inférieur.
- Le cache associatif par ensembles (*Set associative Cache*) : un bloc mémoire peut être copié dans un ensemble de blocs du cache. Les caches associatifs par ensemble ont généralement des ensembles de 2, 4 ou 8 blocs.
- Le cache complètement associatif (*Fully Associative Cache*) : chaque bloc mémoire peut être copié sur n'importe quel bloc du cache.

La complexité de gestion du cache augmente avec son degré d'associativité. En effet, plus le degré d'associativité augmente, plus il faut effectuer une recherche complexe pour savoir si une donnée est présente dans le cache. De même, le coût matériel et la consommation en puissance des caches augmentent avec l'associativité.

Lorsque l'unité de traitement demande une donnée qui n'est pas présente dans le cache, il y a un défaut de cache. Les défauts de cache peuvent être de trois types : les défauts obligatoires de chargement, les défauts de capacité et les défauts de conflit.

- Défauts obligatoires de chargement : ce sont des défauts qui ne peuvent être évités. Le programme accède à des données qui n'ont pas encore été accédées, il faut alors charger le cache pour leur première utilisation.
- Défauts de capacité : le cache ne peut pas contenir toutes les données nécessaires à l'exécution d'un programme. Il faut enlever des données du cache pour les remplacer par de nouvelles données ce qui génère les défauts de capacité.
- Défauts de conflit : ce type de défauts apparaît, pour les caches à adressage direct ou associatif par ensembles, lorsque l'unité de traitement requiert un nombre de blocs dans un même ensemble et dont la taille est supérieure à cet ensemble. En général, le nombre de défauts de conflits est prépondérant par rapport au nombre de défauts de capacités sauf pour les caches complètement associatifs où ils n'existent pas.

Un cache complètement associatif permet d'améliorer les performances en terme d'accès aux données. Cependant l'amélioration de ces performances doit être mise en balance avec le surcoût en matériel et en consommation introduit par la logique de contrôle et de recherche de blocs.

La problématique est que les optimisations "potentielles" obtenues par la présence d'un cache impliquent la mise en œuvre de techniques de remplacement qui sont coûteuses. Nous allons présenter les différentes méthodes de remplacement de données dans les mémoires cache. Nous présentons la gestion matérielle du cache, puis sa gestion logicielle.

c. La gestion matérielle

Le choix des blocs à remplacer dans un cache associatif lors de l'apparition de défauts peut être géré par différents algorithmes de remplacement.

- Le remplacement de l'élément le plus ancien (Last Recently used – LRU) : les dates d'accès aux blocs du cache sont maintenues dans une table. Lorsqu'un défaut apparaît, le bloc ayant la date d'utilisation la plus ancienne est remplacé.
- Le remplacement de l'élément le moins utilisé (Least Frequently Used – LFU) : le taux d'accès aux blocs est conservé et le bloc ayant le taux le plus faible est remplacé lors d'un défaut.

- Le remplacement suivant l'ordre d'ancienneté (First In First Out – FIFO) : le remplacement des blocs s'effectue suivant l'ordre dans lequel ils sont entrés dans le cache.

- Le remplacement aléatoire : les blocs candidats au remplacement sont choisis de manière aléatoire de façon à obtenir une répartition uniforme.

Pour gérer les écritures, le cache doit permettre d'assurer la cohérence des données entre les données contenues dans le cache et celles contenues dans les niveaux inférieurs de la hiérarchie. Deux cas sont à distinguer. Pour le premier, l'écriture a pour cible un emplacement mémoire et possède une copie en mémoire cache. Pour le second, il n'existe pas de copie et l'écriture provoque un défaut. Lorsqu'une copie existe, les caches peuvent avoir les politiques de gestion des écritures suivantes.

- Ecriture simultanée (*Hit Write Through*) : les écritures sont effectuées à la fois en cache et en mémoire, ce qui garantit la cohérence des données entre le cache et la mémoire principale.

- Ecriture différée (*Hit Write Back*) : Les écritures ne sont effectuées que dans le cache. Il n'y a réécriture du bloc modifié en mémoire principale que si celui ci doit être remplacé dans le cache afin de garantir la cohérence.

Dans le second cas, deux politiques de gestion des écritures peuvent être mises en place.

- *Miss fetch on Write* : le bloc dans lequel doit se faire l'écriture est d'abord ramené en mémoire cache avant d'être modifié. Ce mécanisme entraîne une pénalité de défaut d'écriture qui est la même que celle d'un défaut de lecture puisqu'il faut lire un bloc complet depuis le niveau hiérarchique inférieur.

- *Miss write around* : les données sont directement écrites en mémoire principale. Cette technique peut être efficace si le bloc considéré n'est pas réutilisé.

d. La gestion logicielle

Dans un premier temps, tous les caches furent conçus avec une gestion de contrôle matérielle car ils étaient plus simples à concevoir. Cependant, pour des applications dont le code est tout ou partie analysable statiquement, les caches matériels ne sont pas forcément les plus efficaces et sûrement pas les moins consommateurs en terme de puissance. Ce constat a entraîné le développement de caches gérés partiellement de façon logicielle. Les principales différences entre caches à gestion matérielle et caches à gestion logicielle sont les suivantes.

La gestion logicielle du cache est réalisée par le compilateur ou à défaut par le programmeur. Elle nécessite donc moins de ressources matérielles puisque seul le concept de base de transfert de données (de blocs de cache) est nécessaire.

La gestion matérielle réalise les transferts de données en se basant sur l'ordre d'exécution de l'application en utilisant des mesures statistiques fixes. Pour la gestion logicielle du cache, cette tâche est réalisée par le compilateur ou le programmeur. Cela est rendu possible par l'utilisation de directives de compilation.

La plus grande différence réside dans la politique de mise à jour des données, c'est à dire la façon de gérer la cohérence des données. Au niveau matériel, une écriture s'effectue au niveau de hiérarchie mémoire supérieure à chaque apparition d'une écriture ou lorsque qu'un bloc est évincé du cache. En logiciel, c'est le compilateur qui décide à quel moment et si oui ou non une écriture doit être effectuée à un niveau hiérarchique supérieur. Cette gestion permet de réduire le nombre de transferts entre les différents niveaux de la hiérarchie mémoire ce qui entraîne une réduction de la bande passante nécessaire et de la puissance dissipée due aux transferts.

e. Le préchargement des données

Le préchargement des données est une technique d'anticipation d'utilisation et de mise en cache des données. Utilisée dans de nombreux domaines comme dans les caches processeurs [Joup90], les systèmes de stockage [Patt95] ou les systèmes de fichiers [Lei97], le préchargement est particulièrement utilisé en environnement mobile. Il permet en effet de :

- réduire le temps d'accès aux données à l'aide d'un accès en mémoire ou un disque (plus rapide qu'un accès réseau), ce qui est d'autant plus important que la connexion sans-fil possède un faible débit,
- réduire les défauts d'accès aux données lors de déconnexions en permettant de travailler localement sur le cache.

Le succès du préchargement repose sur le fait que le cache doit répondre à la majorité des accès aux données. Pour y précharger les données adéquates, plusieurs stratégies ont été testées dans diffèrents systèmes. Ces stratégies peuvent utiliser des informations fournies par l'utilisateur ou bien être entièrement automatisées et calculer les données à précharger. Ce calcul peut prendre en compte :

- les accès et relations entre données, comme, par exemple, une analyse d'exécution de programme dans [Tait95] ou une distance sémantique entre fichiers dans Seer [Kuen97],
- les relations entre les données et l'environnement, comme les prédictions de mouvements dans [Liu95], [Chim98], [Bhat99].

f. Les mémoires scratch-pads

Les mémoires scratch-pads (ou bloc notes) font référence aux mémoires de données embarquées. Elles ont été introduites par Panda [Pand97a]. La mémoire embarquée est

constituée d'une mémoire cache standard et d'une mémoire SRAM appelée Scratch-pad. Les espaces d'adressage de la scratch-pad et celui de la mémoire externe sont disjoints mais intégrés dans l'espace d'adressage totale du système (voir Figure II.4). Cependant, la mémoire scratch-pad est connectée au même bus d'adresses et au même bus de données. La mémoire cache et la scratch-pad autorisent des accès rapides aux données, mais la scratch-pad garantit un temps d'accès d'un seul cycle alors que la mémoire cache est sujette à défauts. Ce type de mémoire est donc utile pour stocker des données provoquant un grand nombre de défauts de cache.

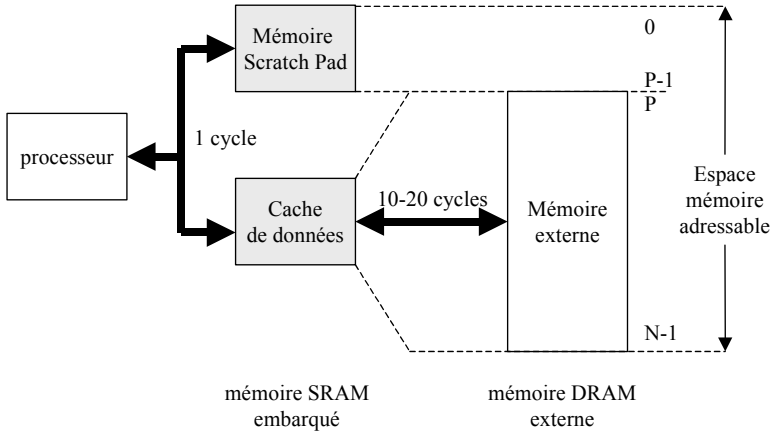


Figure II.4 : Mémoire scratch-pad

Le contenu d'une mémoire scratch-pad est décidé au moment de la compilation et de ce fait est totalement contrôlé par l'utilisateur. A l'inverse, la gestion des transferts de données dans un cache se fait lors de l'exécution. Donc pour des applications dont une analyse complète ne peut être effectuée au moment de la compilation une solution intégrant une mémoire scratch-pad est plus complexe. Pour ces applications, il est préférable d'utiliser un cache "blocable" qui combine les bénéfices d'une mémoire SRAM de type scratch-pad et d'un cache géré matériellement. La partie ne pouvant être analysée de manière statique est gérée par le contrôleur matériel du cache, et la partie pouvant être analysée statiquement est bloquée dans la cache par le compilateur. La gestion des mémoires scratch-pads introduite par [Pand97a] est une gestion statique, et [Avis02] présente un schéma de gestion statique différent pour utiliser des mémoires scratch-pads comme mémoire de données. L'auteur introduit un algorithme de partitionnement des données à travers différentes unités de mémorisation, scratch-pad, DRAM, ROM pour améliorer les performances. Dans les travaux de [Stei02] et [Angi03], des gestions statiques de mémoire scratchpads sont développées pour les mémoires de données et les mémoires d'instructions.

Des techniques de gestion dynamique des mémoires scratch-pads ont récemment été développées ; la gestion dynamique signifie que le contenu des mémoires scratch-pads est modifié durant l'exécution. Cependant, dans la plupart des techniques développées l'analyse

reste statique. Elle a été introduite par [Kand01]. L'architecture mémoire repose sur une mémoire scratch-pad et une mémoire DRAM accédée à travers un cache. Pour transférer les données de la mémoire DRAM à la scratch-pad deux instructions de transfert de données sont créées, une instruction de lecture en DRAM et une instruction d'écriture en scratch-pad. Cette technique a été améliorée par [Uday03] qui propose une gestion globale. L'ensemble d'un programme est analysé pour exploiter la localité globalement et non localement en travaillant uniquement sur chaque nid de boucles. Les travaux réalisés à l'université de Dortmund [Stei02a], [Verm04], intègrent la gestion des instructions. Pour chaque instruction à copier en scratch-pad, une instruction de *load* et une instruction de *store* sont insérées. Le *load* envoie une instruction à copier de la mémoire principale aux processeurs, le *store* la stocke en mémoire scratch-pad. Le code est analysé et les instructions de *load* et de *store* sont insérées dans les parties du code qui sont le plus utilisées.

Dans la plupart des cas, les techniques utilisant des mémoires scratch-pads se limitent à l'étude d'applications pouvant être analysées statiquement. Dans ce cadre d'étude, les performances à base de mémoire scratch-pads sont supérieures à celles des mémoires cache.

g. Conclusion

Les paramètres de fabrication de mémoires cache gérées de façon matérielle ont une influence importante dans la conception d'une architecture à flot de données intensif. Il faut donc définir une architecture et une gestion efficace pour réduire le nombre de défauts et minimiser les transferts de données entre unité de mémorisation et unité de traitement. Par ailleurs il est possible d'améliorer les performances des mémoires en gérant de manière efficace les transferts de données et d'instructions à travers l'architecture et la hiérarchie mémoire.

II.1.3. Exploiter le parallélisme

a. Parallélisme de tâches vs parallélisme de données

La parallélisation est une technique standard utilisée pour améliorer les performances d'un système en utilisant des unités de traitement pouvant opérer simultanément. Cependant, pour une partie de code donnée, les performances du système peuvent énormément varier et il n'apparaît pas de solution simple pour une parallélisation efficace. C'est aussi vrai pour le transfert et le stockage de données. La plupart des recherches dans le domaine des compilateurs s'intéresse au problème de parallélisme et de partitionnement des tâches [Neer93], [Amar95]. En outre, ces techniques n'utilisent que la vitesse comme métrique pour améliorer les performances temporelles ; or la consommation et la taille mémoire ont aussi de l'importance dans les systèmes embarqués, donc des stratégies différentes doivent être développées. Une première

approche pour une optimisation plus globale de la mémoire dans le contexte de processeurs parallèles a été développée à l'IMEC [Mass99]. Une analyse du code permet de réorganiser les boucles avant les phases de parallélisation et de précompilation.

Deux alternatives sont utilisées pour traiter le parallélisme : le traitement du parallélisme de tâches et le traitement du parallélisme des données. Pour le parallélisme de tâches, les différents sous-systèmes d'une application sont assignés à des processeurs différents. Pour le parallélisme de données, chaque processeur exécute l'algorithme entièrement, seulement sur une partie des données. Des solutions hybrides de parallélisation de tâches/données sont aussi possibles. Lorsque l'optimisation du stockage et du transfert de données est un problème, une grande attention doit être portée sur la façon dont l'algorithme va être parallélisé.

b. Illustration d'exploitation du parallélisme

Pour exploiter les accès mémoire parallèles il est nécessaire de définir une bonne organisation de la mémoire et de réduire le nombre d'accès par instruction. Les architectures mémoire classiques de DSP (Digital Signal processor) illustrent l'évolution de l'exploitation du parallélisme avec les architectures Von Neuman ou Harvard (Figure II.5).

Les architectures Von Neuman reposent sur le principe de séquentialisation des accès à une mémoire unique.

Les architectures Harvard reposent sur le principe de découplage entre mémoire d'instructions et mémoire de données. Il est possible de mettre en œuvre plusieurs mémoires de données comme le montre la Figure II.6.

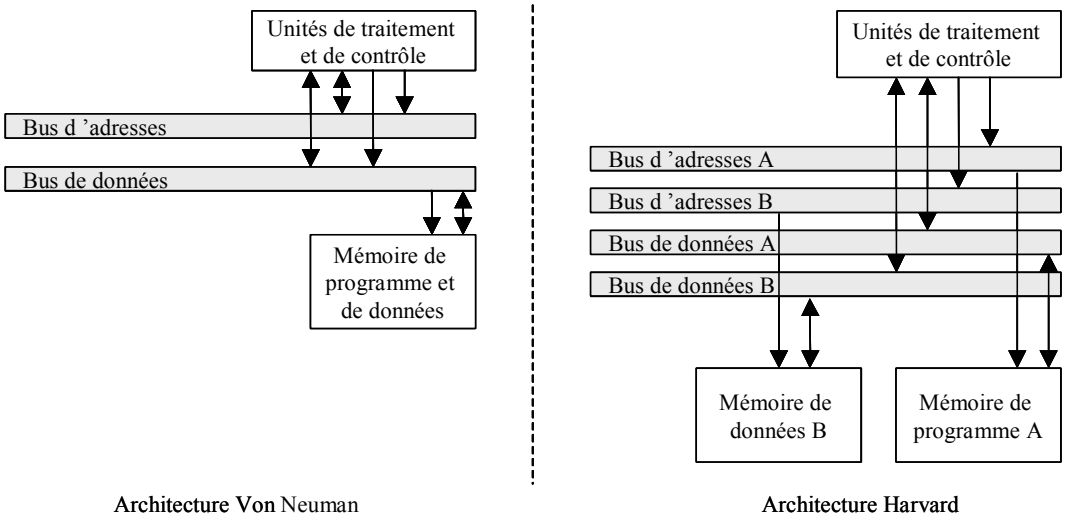


Figure II.5 : Architectures mémoire classiques

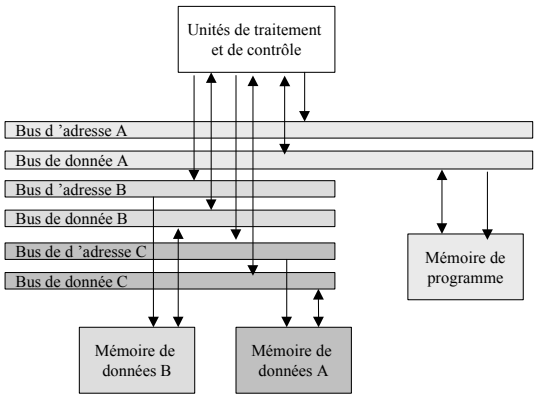


Figure II.6 : Architecture mémoire Harvard à haute performance

Les architectures mémoire pour les DSP à hautes performances intègrent généralement une mémoire de programme et deux mémoires de données. Cela fournit un parallélisme d'accès suffisant pour une unité de traitement généralement pourvue de 2 opérateurs, un additionneur et un multiplieur.

II.2. Exploitation et conception des unités mémoire

Nous présenterons d'abord des solutions développées dans le domaine de la compilation pour réduire les coûts de stockage et de transfert des données à mémoriser. L'étude portera ensuite sur la conception d'unité de mémorisation. Les modèles utilisés pour représenter les accès mémoire seront détaillés avant de présenter des techniques les utilisant pour la conception d'unités de mémorisation. Enfin nous présenterons les alternatives de conception des unités de mémorisation en décrivant les principales étapes et leurs interdépendances.

II.2.1. Exploitation dans le domaine de la compilation

Les compilateurs exploitent de manière optimisée les accès à la mémoire par l'ordonnancement des calculs de l'unité de traitement. L'ordonnancement des calculs doit permettre d'éliminer le maximum d'aléas de donnée ; il y a un aléa de donnée lorsqu'il y a une dépendance de donnée entre différentes instructions et que cette dépendance ne permet pas de modifier l'ordre des instructions. A cause de cette dépendance, l'ordre de la spécification doit être conservé. Le but de la compilation est d'exploiter au mieux le parallélisme en respectant les dépendances de données lorsqu'il affecte les résultats. Les aléas de données possibles sont :

- Une lecture après une écriture. L'opération i essaie de lire une donnée avant qu'elle n'ait été écrite. Il utilise alors l'ancienne valeur. Cet aléa correspond à une vraie dépendance de donnée
- Une écriture après une écriture. Pour ce type d'aléa, l'opération j essaie d'écrire l'opérande avant que l'opération i ne l'écrive. La valeur de destination ne contient pas la

valeur produite par l'opération i mais celle produite par l'opération j. Cet aléa provient d'une anti-dépendance et peut être évité par l'utilisation de l'assignation unique.

- Une écriture après une lecture. L'opération j essaie d'écrire dans une destination avant qu'elle ne soit lue par l'opération i; dans ce cas, i obtient la nouvelle valeur. Cet aléa peut également être évité par l'utilisation de l'assignation unique.

Un ordonnancement dynamique est souvent utilisé pour répondre au problème de gestion des aléas dans les architectures pipeline.

Par ailleurs la distribution des données dans des unités de mémorisation multi-banc soulève le problème d'accès aux données. Lorsque plusieurs bancs mémoires sont disponibles, il faut définir une distribution des données permettant de limiter les conflits d'accès à la mémoire. Pour des architectures de processeur type DSP, la distribution des données en mémoire a un impact important sur les temps d'exécution et la consommation [Senn04].

a. Gestion des accès mémoire par compilation

Le but de la gestion par compilation est d'augmenter le taux de réutilisation des tableaux dans le cache grâce à l'ordre d'exécution et d'éviter les conflits entre tableaux dus à l'ordre de stockage. Ces optimisations vont être réalisées à l'aide de transformation de boucles.

Il existe de nombreuses transformations de boucles pour optimiser les unités de mémorisation dont les principales sont :

- La fusion de boucles (*loop fusion*) : la fusion de boucles consiste à combiner plusieurs boucles en une seule. Cette transformation permet de réduire le nombre de défauts de cache en évitant les accès multiples aux éléments identiques d'une même structure (tableau multi-dimensions). La fusion permet également de réduire les durées de vie des données en mémoire cache en réduisant le temps entre sa production et sa consommation.

- La décomposition de boucle (*loop tiling*) : cette transformation divise l'espace d'itération d'une boucle en bloc ou tuile pour s'assurer que l'ensemble des données permettant de réaliser le calcul est chargé en mémoire cache.

- Le déroulage de boucle (*loop unrolling*) : il consiste à dérouler les boucles totalement ou partiellement, pour réduire le nombre d'accès aux mémoires caches.

Ces différentes transformations visent à augmenter les localités spatiales et temporelles. Elles peuvent être combinées pour réduire les défauts de cache relatifs à l'ordre d'exécution et à l'ordre de stockage. Elles participent à l'amélioration des performances de l'unité de mémorisation en fonction de son architecture et de sa hiérarchie. De nombreuses transformations sont répertoriées et classées dans [Baco94]. Des techniques comme celles développées à l'IMEC intègrent de nombreuses transformations de codes pour exploiter et

optimiser les unités de mémorisation. Cependant l'ordre d'application de ces transformations n'est pas trivial ce qui rend difficile la mise en œuvre de séquence de transformation de code automatique. L'expertise du concepteur reste dans la plupart des cas, la seule solution.

b. La DTSE et Atomium (IMEC)

La méthodologie DTSE (Data Transfert and Storage Exploration) est développée depuis de nombreuses années à l'IMEC. Elle peut se décomposer en deux étapes principales qui peuvent être appliquées à différents niveaux d'abstraction dans le flot de conception. Cette méthodologie est entièrement détaillée dans [Catt98] et [Catt02]. Les actions menées dans cette méthodologie sont représentées sur la Figure II.7.

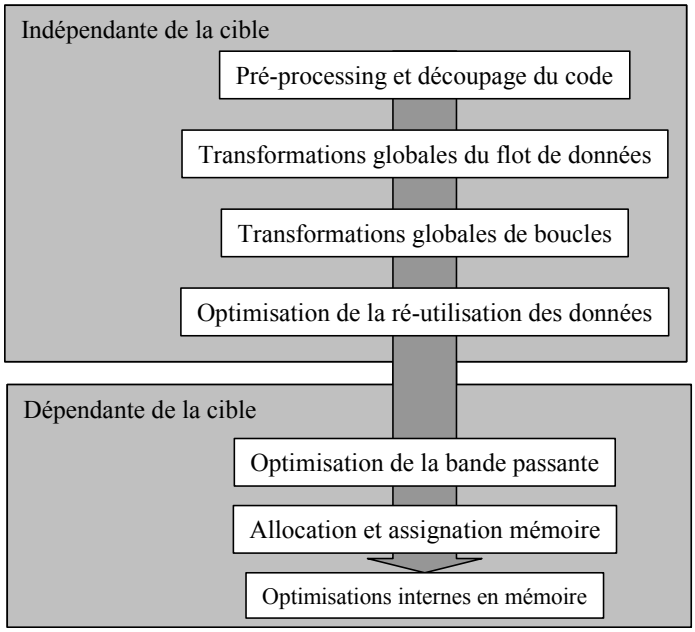


Figure II.7 : Flot de la DTSE

Etape indépendante de la cible.

- Le Pré-processing et le découpage du code sont utilisés pour isoler les transferts de données et les fonctionnalités de stockage d'une application. Cela permet de se focaliser sur les fonctionnalités intéressantes et rend possible l'exploration des alternatives d'optimisation sans tenir compte du style du code utilisé pour spécifier une application.
- Les transformations globales du flot de données [Mass99] sont mises en œuvre pour réduire les transferts de données redondants. Cette étape conduit à un flot de donnée différent sans changer le résultat de la spécification initiale.
- Les transformations globales de boucles [Danc00] introduisent des localités spatiales et temporelles et une régularité globale dans les accès à travers la totalité du code en transformant le flot de contrôle et les structures de boucle du code de l'application.

- L'optimisation de la ré-utilisation des données exploite efficacement la hiérarchie mémoire en introduisant des copies des données les plus souvent ré-utilisées pour réduire la consommation [Digu97]. Cette étape propose le tranfert de données aux niveaux inférieurs de la hiérarchie mémoire pour réduire le nombre de lectures en mémoire principale.

Etape dépendante de la cible.

- L'optimisation de la bande passante des unités de mémorisation est réalisée pour satisfaire les contraintes de temps réel tout en optimisant le coût en consommation en introduisant et en exploitant le parallélisme de transfert des données [Vand99]. Cette étape permet de déterminer la contrainte de bande passante minimale satisfaisant les contraintes temps réel pour une organisation mémoire sous-jacente.
- Une allocation et une assignation mémoire [Sloc97] sont effectuées pour dériver le coût optimisé de l'organisation mémoire en supprimant l'utilisation de mémoire à N-ports. Cette étape est réalisée en générant une organisation mémoire pour un coût minimal (énergie et/ou taille mémoire) et en respectant les contraintes temporelles.
- Les optimisations internes à la mémoire (In-place mapping [Tron02]) peuvent être effectuées en exploitant les durées de vie des variables. Les variables qui ont des durées de vie disjointes peuvent être placées à la même adresse pour réduire les besoins de stockage

L'IMEC a développé un outil appelé ATOMIUM [Imec04](**A** **T**oolbox for **O**ptimising **M**emory **I**/O **U**sing geometrical **M**odels) qui supporte partiellement les étapes de la méthodologie DTSE. Cet outil travaille au niveau comportemental et transforme une application décrite en langage C en une description C équivalente d'un point de vue fonctionnel mais introduisant une réduction du temps d'exécution, de la taille mémoire et de la consommation.

Les applications visées sont des applications multimédia temps réel qui manipulent des tableaux de données. Les cibles de l'outil sont des ASIC (Application Specific Integrated Circuit), des plates-formes à base de microprocesseurs et/ou de FPGA (Field Programmable Gate Array).

II.2.2. Modélisation des accès mémoire

a. Modèles

La représentation des accès mémoire dans la spécification permet d'exploiter et de concevoir des unités de mémorisation adaptées aux contraintes. Le plus souvent les accès mémoire sont modélisés par des nœuds particuliers dans des Graphes Flot de Données (DFG) ou dans des Graphes Flot de Données et de Contrôle (CDFG).

Pour les modèles d'accès mémoire simples, la donnée est lue en mémoire, le calcul est effectué puis la donnée est réécrite en mémoire dans le même cycle d'horloge. Ce modèle est valide pour les registres et des files de registres de petite taille. Cependant, lorsqu'une donnée est mise en mémoire SRAM, le temps d'accès est plus important et devient significatif par rapport au temps de traversée des opérations. Par ailleurs, les mémoires DRAM proposent différents modes d'accès en lecture et en écriture. Il est alors nécessaire d'introduire des modèles d'accès mémoire multi-cycles et de réévaluer les modèles d'ordonnancement des accès aux mémoires supportant des protocoles plus complexes. Dans la littérature, il existe des modélisations pour les accès mémoire à différents niveaux de granularité. Les trois modèles les plus utilisés pour la représentation des accès au niveau algorithmique sont détaillés ci-dessous.

Le premier modèle est une représentation à gros grain. Une opération d'accès mémoire est modélisée par un nœud multi-cycles [Syno97]. Pour les opérations de lecture, la donnée lue est valide seulement à la fin du multi-cycles ; pour les opérations d'écriture, les données doivent être disponibles en début de multi-cycles. La Figure II.8 illustre une représentation à gros grain d'une lecture et d'une écriture multi-cycles. L'axe horizontal représente le temps, il est décomposé en cycles (traits verticaux). Pour une lecture multi-cycles, l'adresse doit être présente au premier cycle de l'opération multi-cycles, la donnée est disponible à la fin du dernier cycle. Pour l'écriture la donnée et l'adresse doivent être présentes au premier cycle de l'opération multi-cycles.

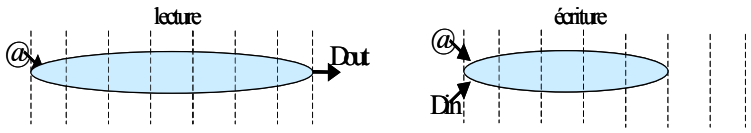


Figure II.8 : Représentation des accès mémoire à gros grain

La seconde modélisation développée repose sur une décomposition à grain moyen. Les accès à la mémoire sont représentés par des nœuds indépendants encapsulés dans un nœud global, c'est la notion de "*Behavioral template*" développée par [Knap95]. Ces nœuds peuvent être multi-cycles ou non.

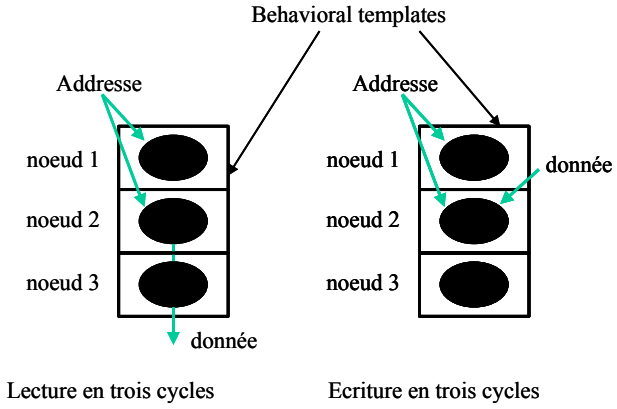


Figure II.9 : Modélisation à grain moyen

Le dernier modèle d'accès mémoire est un modèle à grain fin qui permet de modéliser tous les modes d'accès d'une mémoire DRAM. Un premier modèle permet de représenter les accès mémoire à l'aide de trois nœuds distincts. Un nœud de *décodage de ligne*, un nœud de *décodage des colonnes* (pour les lectures ou les écritures) et un nœud de *préchargement*. La Figure II.10 illustre la modélisation des lectures et des écritures avec ce modèle.

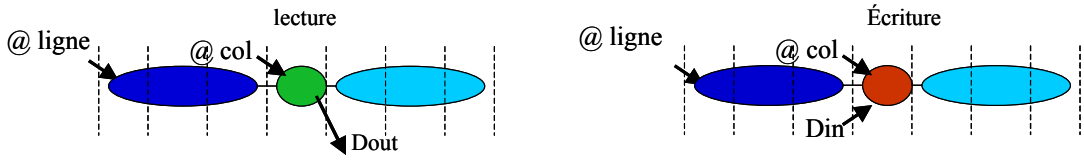


Figure II.10 : modélisation des lectures et écritures à grain fin

Ce modèle a été complété par un nœud *Setup* (Figure II.11). Il permet le chaînage d'opérations de lectures et d'écriture et autorise ainsi la modélisation du mode d'accès mémoire Read-Modify-Write (RMW). Il est utilisé dans [Pand98] pour réaliser des transformations de graphes permettant d'exploiter au mieux les possibilités des modes d'accès offerts par les mémoires DRAM ; cela permet d'optimiser le nombre d'accès et de réduire la latence pour une spécification algorithmique donnée.

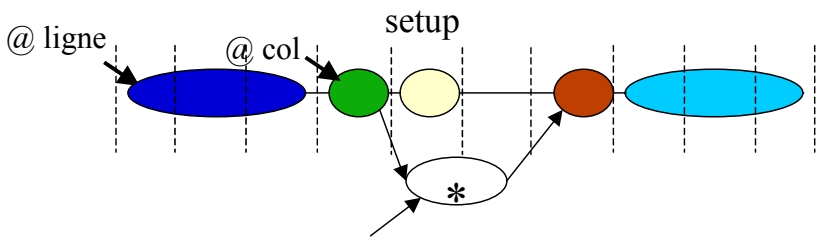


Figure II.11 : Représentation mode d'accès Read Modify Write

Ces différents modèles permettent de représenter, à différents niveaux de granularité, les accès mémoire dans des graphes flot de donnée ou flot de donnée et de contrôle. Ils sont utilisés pour exploiter les possibilités d'optimisation offertes par les modes d'accès des mémoires au niveau algorithmique ou architectural. Nous présenterons le modèle que nous utilisons pour représenter les accès mémoire et justifierons notre choix dans le chapitre IV.

b. Exemple d'exploitation des modèles

Les principaux travaux concernant la gestion des unités de mémorisation pour la synthèse de haut niveau à l'Université de Californie, Irvine, ont été menés à la fin des années 90 par P.R. Panda [Pand98], [Pand99a], [Pand99b]. L'auteur aborde le problème de l'exploration et de l'optimisation de la mémoire pour les systèmes embarqués. Il suppose que le partitionnement logiciel/matériel a été réalisé par le concepteur ou par une étape de conception amont (Figure

II.12). Pour les solutions matérielles, il traite du problème d'interfaçage entre les blocs matériels synthétisés et les mémoires externes en orientant son étude vers un flot de synthèse de haut niveau. Pour les solutions logicielles, il s'intéresse à l'interfaçage entre l'unité de traitement, la mémoire interne et le mémoire externe en introduisant des techniques exploitant les mémoires scratch-pads.

Solution matérielle.

Le modèle d'accès mémoire est le modèle à grain fin présenté dans la section II.2.2. Ce modèle permet de représenter les modes d'accès aux mémoires DRAM dans un CDFG. L'intégration de ce modèle permet d'exploiter les modes d'accès par pages des mémoires DRAM lors de l'ordonnancement. L'exploitation des accès mémoire DRAM s'effectue par des transformations du CDFG. Ces transformations permettent de maximiser le nombre d'accès par page en exploitant les modes d'accès au mémoire DRAM tels que (Lecture, Modification, Ecriture). Les transformations de boucles appliquées, une dernière étape consiste à assigner les structures de données aux mémoires externes. Durant cette étape, les données structurées sont d'abord assignées à des mémoires virtuelles, puis à des mémoires physiques.

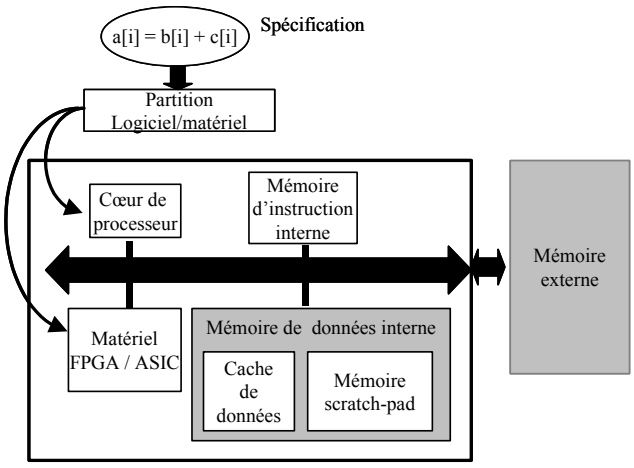


Figure II.12 : Flot de synthèse à base de processeur embarqué

Solution logicielle.

L'auteur développe une stratégie de placement des données en mémoire cache pour gérer au mieux l'interaction entre le processeur et la mémoire externe. En s'appuyant sur la construction de graphes de localités et de conflits d'accès aux variables, l'auteur détermine une organisation mémoire intégrant un cache de données. Cette stratégie suppose que l'on connaisse a priori les séquences d'accès aux données et n'est applicable que pour des applications déterministes. Dans ces travaux, l'auteur s'est également intéressé à l'apport de mémoires scratch-pad et au partitionnement des données entre mémoire cache et scratch-pad pour des applications déterministes [Pand97a].

II.2.3. Conception optimisée

La conception optimisée d'unité de mémorisation, dans le cadre restreint des applications déterministes, peut être envisagée en plusieurs étapes interdépendantes. Ces étapes sont la distribution des structures de données, le placement des structures de données, l'ordonnancement des opérations de l'unité de traitement et la génération des adresses.

- L'ordonnancement fixe les séquences d'accès aux données. Il impose donc des contraintes fortes sur la distribution et le placement des données dans l'architecture de l'unité de mémorisation.
- La distribution des données a une grande influence sur le placement. Elle assure la cohérence des données et exprime le parallélisme qui pourra être exploité dans l'ordonnancement.
- Le placement des données en mémoire consiste à attribuer une adresse à chaque donnée distribuée en mémoire. L'attribution d'une même adresse pour plusieurs données permet de limiter la taille globale de la mémoire.
- La génération d'adresses nécessite la mise en œuvre de matériel fournissant les adresses des données à accéder. La génération des adresses impose un placement et une distribution et contraint l'ordonnancement. Elle peut s'effectuer durant les phases de distribution ou de placement de manière à obtenir une unité mémoire optimisée.

La difficulté est de déterminer l'ordre de ces étapes permettant une conception optimisée de la mémoire. La Figure II.13 fait apparaître le cercle de contraintes des optimisations pouvant être réalisées. Les flèches pleines représentent des contraintes fortes d'une étape d'optimisation à l'autre, les flèches pointillées représentent des contraintes plus souples.

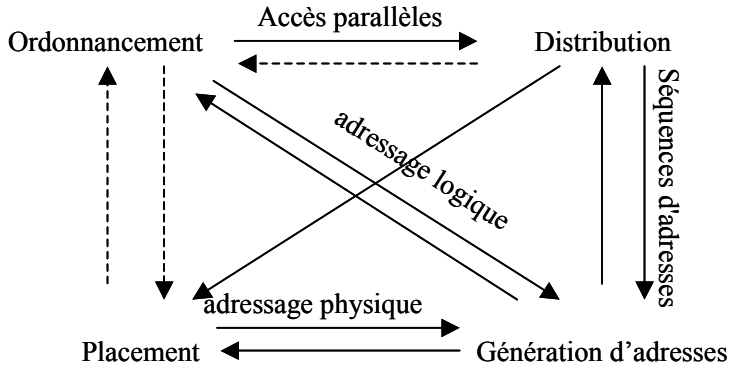


Figure II.13 : Cercle de contraintes des optimisations

Chaque phase a une influence sur les autres. Il semble alors nécessaire de développer une approche de conception optimisée globale. Cependant, une approche globale est difficile à mettre en place car cela entraîne une complexité pour l'optimisation très forte. Il faut donc trouver un enchaînement des phases de conception de la mémoire qui conduit à une solution

optimisée. Nous verrons comment nous plaçons notre stratégie de conception optimisée dans ce cercle de contraintes mémoire lors de la définition de notre cahier des charges à la fin de ce chapitre.

II.3. La synthèse de haut niveau

La synthèse de haut niveau a été développée pour réduire le temps de conception de solutions matérielles en élevant le niveau d'abstraction. Elle suit une démarche similaire à la compilation avec en supplément l'objectif de produire une description au niveau RTL (Register Transfert Level) des unités architecturales qui répondent aux contraintes fonctionnelles et temporelles d'une application. La spécification d'entrée est une description comportementale en langage de haut niveau orienté vers le matériel (VHDL (Very high speed integrated circuit Hardware Description Language), Verilog, SystemC, …). L'automatisation du flot de synthèse haut niveau permet de réduire considérablement le temps entre la spécification et le silicium et d'augmenter la fiabilité de conception.

Le flot typique de synthèse de haut niveau produit une description structurelle d'une architecture en transformant, par un ensemble d'étapes successives, la description initiale de l'algorithme. Ainsi, l'étape de **compilation** réalise la vérification syntaxique et sémantique de la description source et la traduit en un format intermédiaire propre à l'environnement de synthèse. La phase de compilation réalise des opérations, que l'on retrouvent dans les compilateurs ILP (Integer Level Parallelism), telles que : l'élimination du code mort, la propagation des expressions constantes, le déroulage de boucles, la mise en ligne des fonctions. Deux classes de modèles sont couramment employés pour la **représentation intermédiaire**, les *DFG* et les *CDFG*. Le premier modèle permet d'exhiber les dépendances de données et d'exprimer le parallélisme maximum de calcul des applications. Le CDFG qui sont plus adaptés aux applications orientées contrôle.

L'étape de **sélection** consiste à choisir la nature des ressources matérielles (opérateurs) qui réaliseront les opérations. Le choix des composants se fait sur des critères tels que la surface, la vitesse ou la consommation.

L'étape **d'allocation** détermine, pour chaque type d'opérateur sélectionné, le nombre de ressources utilisées dans l'architecture finale.

L'étape **d'ordonnancement** affecte une date d'exécution à chacune des opérations en tenant compte d'une part des dépendances de données et d'autre part des contraintes. Ainsi, l'ordonnancement peut chercher à :

- minimiser le nombre d'étapes de contrôle en fonction d'une quantité de ressources *(List Scheduling...)*

- minimiser le nombre de ressources en fonction d'un nombre d'étapes de contrôle *(Force Directed Scheduling...)*
- minimiser le nombre de ressources et le nombre d'étapes de contrôle *(Force Directed List Scheduling...)*

L'étape **d'assignation** associe à chaque opération un opérateur. Il est important de noter que l'ordre des étapes de synthèse peut varier selon les outils et les contraintes supportées. En effet, certaines techniques de synthèse effectuent l'ordonnancement puis l'assignation, d'autres l'assignation puis l'ordonnancement et dans certains cas l'ordonnancement et l'assignation de façon simultanée.

Ces étapes permettent d'obtenir une description des différentes unités fonctionnelles au niveau RTL orientées vers des solutions ASIC ou FPGA. Toutes les phases du flot de synthèse d'architecture ont une influence sur la gestion mémoire. Si aucune précaution n'est prise au cours de la synthèse, les solutions architecturales pour les unités de mémorisation peuvent s'avérer très coûteuses. Il apparaît alors nécessaire d'intégrer la gestion de la mémorisation dans le flot de synthèse d'architecture pour garantir les performances de l'application et aboutir à des solutions architecturales réalistes et optimisées en terme de consommation, vitesse et surface.

Nous présentons, dans les paragraphes suivants, les méthodes développées dans la littérature pour la conception à haut niveau d'unité de mémorisation. De façon générale, de nombreuses techniques ont été développées pour réguler le flot de données entre l'unité de traitement et l'unité de mémorisation d'une part et réduire la bande passante mémoire d'autre part.

II.3.1. Distribution

a. Distribution de scalaires en mémoire simple et multiports

Les premières techniques gèrent la mémorisation à l'aide de registres ; elles supposent généralement que la phase d'ordonnancement est effectuée. Le problème relatif à la gestion de mémorisation par registres, appelé assignation de registres, consiste à utiliser aux mieux les registres afin de minimiser le nombre de registres assignés aux variables. La réduction du nombre de registres permet de limiter fortement le coût en surface et en consommation. Ces techniques sont cependant limitées par la forte complexité de l'optimisation. Le coût de mémorisation en registre reste important par rapport à celui d'une mémoire lorsque le nombre de variables à mémoriser est important.

Les registres ont donc été remplacés par des modules mémoires et de nouveaux problèmes d'optimisation se sont posés. Les données ne peuvent être accédés simultanément car le nombre d'accès simultanés est limité par le nombre de ports disponibles de la mémoire. Cela implique une forte corrélation entre la décision d'allocation mémoire et la phase d'ordonnancement de la

synthèse de haut niveau. Des techniques pour allouer des mémoires multiports lors de la synthèse de haut niveau ont été développées par [Bala88]. Pour exploiter le regroupement de plusieurs registres dans une seule mémoire multiports, ces techniques tentent de fusionner les registres ayant des temps d'accès disjoints. Le partitionnement en clique est suffisant pour s'occuper du cas de mémoires simple port, mais une méthode plus générale est nécessaire pour traiter les mémoires multiports. Le problème de l'assignation de mémoires multiports est modélisé par un modèle de type *Integer Linear Programming (ILP)* intégrant le type de ports (*read, write, read/write*), le nombre de ports, et l'ordonnancement des accès aux registres à chaque étape de contrôle. Puisque le problème est NP-complet, un algorithme *Branch and bound* est utilisé. L'utilisation de cette heuristique n'aboutit pas à un nombre minimum de modules mémoires. Pour minimiser le nombre total de mémoires multiports, le problème ILP doit être généralisé en ajoutant les modèles des mémoires utilisées dans la formulation.

Les travaux [Kram92] définissent un compromis entre distribution en modules mémoire SRAM et distribution en registres pour les données structurées de spécifications comportementales décrites dans plusieurs process VHDL. Un compromis performances/surface induit par les accès est réalisé lors de la génération des clusters de registres (regroupement de plusieurs registres). L'allocation des variables scalaires aux files de registres ou aux modules mémoires permet de réduire le coût intrinsèque (couple vitesse/surface) d'un circuit.

Dans la technique développée par [Kim93], l'auteur cherche d'abord à réduire le coût des interconnexions entre l'unité de traitement et les éléments de mémorisation avant de s'intéresser aux phases d'allocation/assignation mémoire. [Lee95] présente une technique pour effectuer l'allocation et l'assignation des variables en registres en même temps que l'ordonnancement. L'assignation et l'ordonnancement sont réalisées simultanément en se basant sur des listes de priorités déterminées sur les accès mémoire pour tenter d'harmoniser le nombre de transferts de données.

b. Distribution des données structurées en mémoire

Dans une organisation mémoire sur mesure, le concepteur peut choisir les paramètres mémoire tels que le nombre de mémoires, la taille, le nombre de ports de chaque mémoire. Ces décisions influencent la distribution des données structurées en mémoire.

De nombreuses techniques s'intéressent au problème de la distribution des données structurées d'une spécification dans une architecture mémoire donnée. Les premières ont été développées dans le contexte des FPGA [Karc94]. Les données structurées sont représentées par des mémoires logiques (taille des tableaux), qui vont être distribuées sur les différentes mémoires physiques disponibles dans l'architecture.

Le problème de la distribution des données est souvent traité après l'étape d'ordonnancement lorsque les séquences accès aux données structurées sont connues. La méthode la plus courante pour réaliser la distribution des données structurées repose sur la construction de graphes de conflit où les nœuds représentent les données structurées et les arcs représentent l'impossibilité de distribuer des données structurées dans la même mémoire.

L'allocation de plus ou moins de mémoires a un effet important sur la surface et la consommation de l'architecture mémoire. Les grandes mémoires consomment plus d'énergie par accès que les petites, à cause de la longueur des mots et des bit-lines [Kamb97], [Joup99]. Donc, l'énergie consommée par une grande mémoire contenant toutes les données sera plus élevée que l'énergie consommée pour une architecture où les données sont distribuées sur des mémoires plus petites. De plus ces petites mémoires peuvent être mises en veille ou sous tension d'alimentation réduite pour limiter la consommation lorsque les données stockées ne sont pas accédées. En outre, la surface de la solution à une seule mémoire est souvent plus importante lorsque les données ont des tailles différentes. A l'opposé, distribuer chaque donnée structurée (tableau) dans une mémoire différente conduit aussi à une consommation d'énergie relativement importante à cause de la multiplication des interconnexions entre les différentes mémoires et les chemins de données. Il apparaît clairement qu'il existe un compromis entre ces deux extrêmes. Dans ces travaux, [Broc00] montre que les coûts, en terme de compromis surface-consommation, évoluent suivant une courbe en forme de "cuvette". Cette courbe permet d'extraire le nombre de mémoire à allouer pour obtenir un coût minimum.

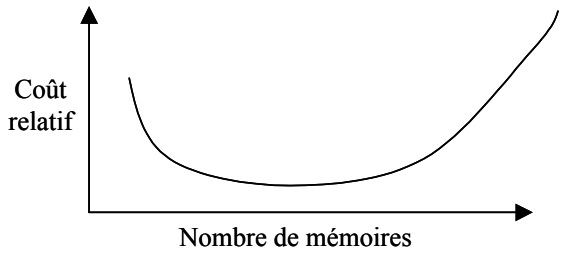


Figure II.14 : compromis coût / allocation mémoires (source [Broc00])

II.3.2. Ordonnancement

Dans de nombreux cas, une architecture complètement sur mesure fournit une bande passante mémoire et des caractéristiques de consommation plus intéressantes qu'une architecture de hiérarchie mémoire traditionnelle incluant un cache de données. Cela est particulièrement vrai quand la compilation permet de connaître les paramètres d'une application (le contrôle dépend uniquement de valeurs connues au moment de la compilation). Concevoir une architecture mémoire spécifique signifie décider combien de mémoires utiliser et quels

types de mémoires utiliser. De plus, les accès mémoires doivent être ordonnés dans le temps de façon à tenir les contraintes temps réel.

Un des facteurs importants qui affecte le coût d'une architecture mémoire est l'ordre relatif des accès mémoire contenus dans la spécification. Des techniques pour optimiser le nombre de ressources, comme les ordonnancements de type *force directed scheduling*, donnent des résultats efficaces mais se limitent à la manipulation de scalaires. La plupart des premières méthodes d'ordonnancement prenant en compte la partie mémoire essaient de réduire le coût mémoire en estimant les besoins en terme de nombre de registres pour un ordonnancement donné (seulement pour les scalaires). Les quelques exceptions sont les *stream schedulers* [Verh95], les *rotation schedulers* [Pass95], les *percolation schedulers* [Nico93]. Ils ordonnancent les accès et les opérations (dans un contexte de compilation) en évaluant l'impact des accès mémoires sur les performances. Peu d'entre eux essaient de réduire la bande passante mémoire ; ils le font en minimisant le nombres d'accès simultanés [Verh95], mais ils ne tiennent pas compte de quelles données sont accédées en parallèle. Par conséquent, aucun effort significatif n'est fait pour optimiser les graphes de conflits d'accès aux données pour effectuer dans de meilleures conditions la tâche postérieure d'assignation de registres et mémoires.

La liberté d'ordonnancement autour des accès mémoire peut aussi être exploitée pour générer des architectures mémoires avec des coûts plus faibles (nombre de mémoires) et des bandes passantes réduites (nombre de ports). Des techniques d'optimisation de la bande passante, de mémorisation et de distribution uniforme des accès mémoire sont développées à l'IMEC [Wuyt99].

II.4. Les outils de synthèse de haut niveau

De nombreux outils de synthèse de haut niveau ont été développés au cours des quinze dernières années. Beaucoup d'entre eux se focalisent sur la synthèse de l'unité de traitement. Nous présentons, dans cette section, uniquement les outils de synthèse qui intègrent la conception des unités de mémorisation dans leur flot.

II.4.1. SystemC compiler / CatapultC

Behavioral Compiler de Synopsys [Syno97] et Monet de Mentor Graphics [Ment98] sont les premiers outils commerciaux de synthèse d'architecture ayant vu le jour. Ces outils ont respectivement été remplacés par SystemC Compiler [Syno03] et Catapult C [Ment04]. Bien que n'ayant pas de domaine d'application ciblé, ils reposent tous deux sur un flot de synthèse qui privilégie les algorithmes de calcul intensif. La description comportementale initiale peut être réalisée à l'aide des langages systemC pour Synopsys et C++ pour Mentor Graphics. Elle est transformée, lors de la compilation, en une représentation interne de type CDFG. Le

comportement de l'algorithme est décrit sous la forme d'un unique processus dans lequel les échanges de données avec l'extérieur sont réalisés au travers de ports. La synchronisation est spécifiée directement dans le code source au moyen d'instructions *wait* associées à la détection d'un front d'horloge. Ils offrent deux modes d'ordonnancement qui se différencient par leur gestion du comportement aux entrées/sorties et/ou le nombre de cycles de contrôle qu'ils peuvent insérer entre deux états d'attente [Ly95].

- Le mode *cycle-fixed* est le plus contraint et permet de maintenir des comportements pré et post synthèse identiques en préservant l'ordre et les relations temporelles sur les opérations d'E/S. L'ordonnancement des calculs s'effectue donc dans les limites fixées par les contraintes sur les entrées/sorties qui, dans certains cas, peuvent rendre la synthèse impossible.
- Le mode *Superstate-fixed*, à la différence de son prédécesseur, conserve uniquement une relation d'ordre d'acquisition et de production des E/S. Dans ce contexte, deux instructions d'attente *wait* adjacentes sont considérées comme les bornes d'un état spécifique nommé *super-état*. Les opérations contenues dans un super-état ne sont plus ordonnancées dans un nombre de cycles fixé par la spécification puisque l'ordonnancement s'autorise un décalage temporel sur les entrées/sorties en étendant la durée d'un super-état par ajout de cycles d'horloge. Les écritures des sorties sont réalisées simultanément sur le dernier front d'horloge.

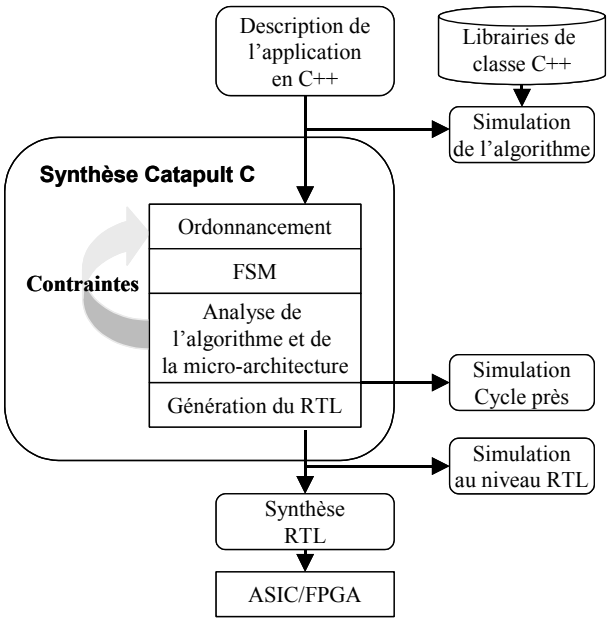


Figure II.15 : Flot de synthèse de l'outil Catapult C

Le concepteur peut utiliser, en plus du mode d'ordonnancement, un ensemble de contraintes temporelles spécifiées à l'aide de directives de synthèse. Ces contraintes spécifient des délais

relatifs minimum et maximum entre deux instructions marquées explicitement dans le code à l'aide de *tags* [Ly95].

Les mémoires sont traitées comme des composants "comportementaux", ce qui veut dire que les composants relatifs à la mémoire doivent être spécifiés comme des composants séquentiels dans une librairie de composants. Avec ce traitement, l'utilisateur peut instancier des composants qui correspondent aux mémoires contenues dans une netlist. Les mémoires pouvant être instanciées sont des mémoires RAM ou ROM, et l'outil Catapult C permet par ailleurs l'instantiation de FIFO. Pour exprimer une opération de lecture ou d'écriture en mémoire, il faut utiliser les deux opérateurs pré-définis en librairies (*read_mem* et *write_mem*) qui sont considérés comme des opérateurs d'entrée/sortie. Pour chaque mémoire, différents types de ports peuvent être utilisés (multi-ports, accès en lecture seule, accès en écriture seule, accès en lecture/écriture). L'instantiation d'une mémoire correspond à une assignation de tableau à une RAM, ROM ou FIFO. Les accès mémoire sont considérés comme des opérations multi-cycle (opérateurs *read_mem* et *write_mem*) dans le Graphe de Contrôle et de Donnée *(CDFG)*. Ils sont représentés par des nœuds comportementaux (*behavioral template*) qui encapsulent plusieurs nœuds du CDFG et fixent leurs ordonnancements relatifs [Knap95]. Les nœuds mémoire (*behavioral template*) sont ordonnancés comme les nœuds opérations de l'unité de traitement. Les conflits d'accès aux données sont gérés par la possibilité ou non d'ordonnancer les opérateurs de lecture et d'écriture *read_mem* et *write_mem* alloués à chaque mémoire. En pratique, le nombre de nœuds de la spécification d'entrée doit être limité pour obtenir une solution architecturale réaliste et satisfaisante à cause de la complexité de l'algorithme utilisé. Les architecture mémoires sont des architectures à bases de mémoire SRAM, l'allocation et le placement des données sont réalisés par le concepteur qui peut placer une ou plusieurs données structurées dans une même mémoire. L'architecture de l'unité de traitement peut être parallèle dans la mesure ou le parallèlisme d'accès aux données n'est pas un facteur limitant. Les unités de traitement ne peuvent pas être pipeline. Le problème de la gestion des accès mémoires pour des architectures pipelines n'est pas traité.

II.4.2. GAUM, synthèse mémoire en aval de l'outil GAUT

Une première approche de conception d'unité de mémorisation autour de la synthèse de haut niveau a été développée dans les travaux de thèse de Daniel Chillet [Chil97]. Le projet GAUT s'est primitivement développé pour fournir une description architecturale des unités de traitement. L'ensemble des informations fournies par la synthèse de l'unité de traitement propage ses contraintes vers la synthèse des autres unités fonctionnelles. Dans ces travaux, l'auteur propose une méthode et un outil s'insérant en aval de la synthèse des unités de traitement. Il

présente une méthodologie visant à aider le concepteur en proposant une analyse des besoins de mémorisation.

L'auteur dégage trois phases principales de synthèse pour les unités mémoire ainsi que leur ordre d'application :

- la distribution des données dans les bancs mémoire
- le placement des données dans chaque banc mémoire
- la génération des adresses

La phase de distribution commence par une analyse des séquences de transferts (lecture et écriture) entre unité de traitement et unité de mémorisation. Le nombre de transferts simultanés entre les deux unités est observé après l'ordonnancement des opérations dans l'unité de traitement. Pour réduire le nombre d'accès parallèles, les séquences de transferts de données sont remises en cause. Une analyse du nombre d'accès par cycle de contrôle et des dépendances de données permet d'anticiper les lectures et de retarder les écritures de façon à uniformiser le nombre d'accès par cycle. Le nombre de mémoires à utiliser est déterminé afin de garantir la cohérence des données et le parallélisme d'accès. La distribution est réalisée en affectant un numéro de banc mémoire à chaque donnée.

La phase de placement des données consiste à affecter une adresse physique à chacune des données. Le placement doit tenir compte de la complexité des générateurs d'adresses qui vont être associés à chaque banc mémoire. La phase de génération des adresses est couplée au placement des données. L'auteur met en œuvre des stratégies de placement des données pour différentes solutions architecturales de générateurs d'adresses. Les générateurs d'adresses peuvent être des machines d'états, des compteurs/décompteurs, des UAL avec registres de base et d'index ou des compteurs+mémoires ROM. L'auteur définit une stratégie de placement en fonction des générateurs d'adresses sélectionnés et du type de données manipulées. Les solutions retenues tentent de minimiser le coût de la solution architecturale de l'unité de mémorisation en fusionnant les générateurs d'adresses.

La stratégie développée permet de gérer les accès aux données scalaires uniquement. Les performances de l'unité de traitement peuvent entraîner un nombre important d'accès simultanés à la mémoire. Les accès mémoires vont être distribués dans le temps de manière à réduire la complexité de l'unité de mémorisation (réduire le nombre de ports et le nombre de mémoires.

II.5. Conclusion

Dans le cadre du domaine d'application du traitement du signal et de la synthèse de haut niveau, l'enjeu du travail repose sur la mise en place d'une méthodologie de conception et de gestion des unités de mémorisation. Cette méthodologie devra permettre la conception de

solutions architecturales efficaces pour les unités de mémorisation et pour les unités de traitement.

Nous présentons notre stratégie de synthèse et son positionnement par rapport aux stratégies développées dans les outils de Synopsys et Mentor Graphic et dans l'outil GAUM.

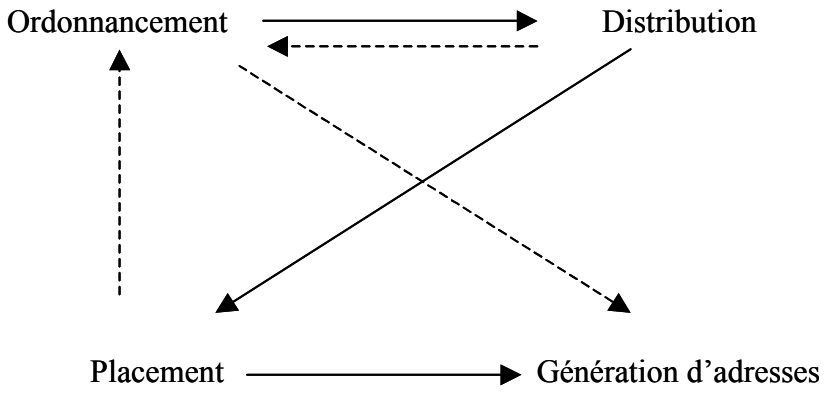


Figure II.16 : Stratégies de synthèse

La Figure II.16 reprend la Figure II.13 décrivant le cercle de contraintes d'optimisation. La stratégie développée dans l'outil GAUM est représentée par les flèches pleines. La stratégie développée dans les outils SystemC compiler et CatapultC est représentée par les flèches pointillées. Nous choisissons le même ordre de conception que celui utilisé dans les outils SystemC compiler et CatapultC en utilisant des méthodes innovantes pour traiter la phase d'ordonnancement sous contraintes mémoire. Ces méthodes ainsi que leurs implantations dans notre outil de synthèse nous permettra de réduire le temps d'obtention d'une description RTL à partir d'une spécification algorithmique et de contraintes de mémorisation.

	GAUM	System C compiler	CatapultC	Approche proposée
Ordonnancement	Avant distribution et placement des données	Sous contraintes de distribution et de placement	Sous contraintes de distribution et de placement	Sous contraintes de distribution et de placement
Granularité	Scalaires	Données structurées	Données structurées	Données structurées ou scalaires
Architectures parallèles	OUI	OUI	OUI	OUI
Architectures pipelines	OUI	NON	NON	OUI

Tableau II-3 : Comparaisons des stratégies de synthèse

Le Tableau II-3 caractérise les stratégies de synthèse des approches commerciales, l'approche GAUM et l'approche que nous proposons. Pour pouvoir profiter des opportunités d'optimisation offertes par les données manipulées dans les applications de traitement de l'image et du signal, nous traiterons les différents types de données de manière différente. Les outils commerciaux SystemC compiler et CatapultC ne permettent pas d'intégrer des architectures de mémorisation efficaces suivant les types de données. Au niveau de l'ordonnancement des opérations de l'unité de traitement, nous intégrerons des contraintes de distribution et de placement des données. L'introduction de ces contraintes permet de maîtriser l'architecture de l'unité de mémorisation. L'approche GAUM consistant à déterminer la distribution et le placement des données ainsi que l'architecture mémoire après synthèse de l'unité de traitement est moins flexible et plus complexe.

La stratégie mise en œuvre fait apparaître deux phases. Dans une première phase, nous cherchons à limiter la complexité du problème de la distribution et du placement des données en manipulant des données structurées. Cette phase est située en amont de l'ordonnancement des opérations de l'unité de traitement. La seconde phase répond au problème de données spécifiques aux applications TDSI. Nous développons une gestion "fine" des données critiques en manipulant des scalaires.

Notre approche propose (1) de définir la distribution et le placement de données structurées en amont de l'ordonnancement ; (2) de réduire la complexité de l'ordonnancement sous contraintes de mémorisation (contraintes de distribution et de placement des données en mémoire) ; (3) d'adapter les architectures de l'unité de mémorisation aux caractéristiques des données des applications TDSI.

Chapitre III
Stratégie globale de gestion de la mémorisation

Dans ce chapitre, nous présenterons la stratégie globale de gestion de la mémorisation. Cette stratégie a pour but de répondre à des besoins fonctionnels en prenant en compte diverses contraintes applicatives et technologiques (débits, cibles technologiques, mémoire ….). Dans un premier temps nous présenterons les solutions proposées en amont de la synthèse d'architecture pour le matériel ou de la compilation pour le logiciel. Nous introduirons ensuite notre outil de synthèse GAUT et les principales caractéristiques nous permettant de définir une approche visant à intégrer la gestion de la mémorisation dans le flot de synthèse d'architecture.

III.1. Approche de conception

III.1.1. Stratégie de conception

La stratégie développée au sein du projet faible consommation s'inscrit dans une démarche globale d'optimisation de la mémoire tant au niveau logiciel que matériel (Figure III.1). Le problème de conception et de gestion des unités de mémorisation est décomposé en deux phases de granularité différentes. Pour simplifier la gestion de la mémorisation des données, le problème est d'abord traité à gros grain en amont de la synthèse ou de la compilation (manipulation de données structurées). La manipulation de scalaires permet ensuite de raffiner la distribution et le placement lors de la synthèse de haut niveau. La stratégie de conception matérielle intègre la prise en compte des ressources de mémorisation au sein de la synthèse automatique d'architectures de traitement réalisée par l'outil de synthèse *GAUT*.

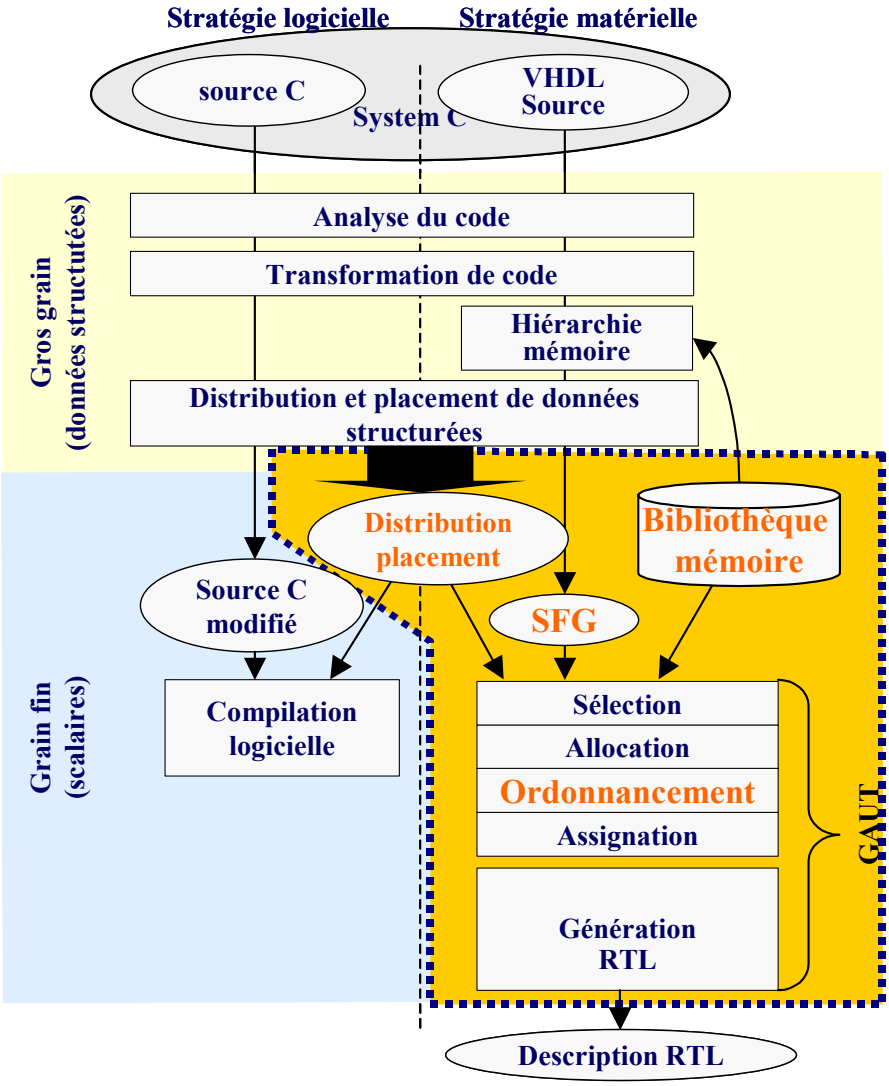


Figure III.1 : Stratégie d'intégration de la mémoire autour de *GAUT*

La première phase, en amont de la synthèse d'architecture, explore la hiérarchie et l'architecture mémoire. Une distribution des données structurées est réalisée à partir d'une spécification algorithmique. Elle permet de réaliser une optimisation à gros grain (données structurées) sur la distribution) et de définir les contraintes de mémorisation qui vont être intégrées au processus de synthèse. Ce travail est actuellement l'objet des travaux de thèse de Florian Marteil [Mart04]. La phase en aval de la synthèse d'architecture consiste à placer les données scalaires (grain fin) générées lors de la synthèse dans l'architecture mémoire existante et à concevoir les générateurs d'adresses de l'unité de mémorisation. Une approche consistant à développer une architecture mémoire, un placement et des générateurs d'adresses en aval de la synthèse d'architecture a été intégrée dans l'outil GAUM [Chil97].

Notre travail s'inscrit dans une phase intermédiaire et considère un ensemble de données structurées, distribuées et placées (noté mapping mémoire sur la Figure III.1) et un ensemble de composants mémoire disponibles pour une cible technologique donnée (bibliothèque mémoire sur la Figure III.1), qui seront de nouvelles contraintes en entrée de la synthèse de haut niveau.

III.1.2. Détermination de la hiérarchie et de la distribution des structures

La première phase se situe en amont de la synthèse d'architecture et de la compilation ; elle comporte des méthodes indépendantes de l'architecture ciblée. Elle commence par l'analyse du code source afin de modéliser l'application sous la forme d'un graphe de structure. Des transformations de code permettent d'optimiser le nombre de transferts de données. On définit ensuite la hiérarchie mémoire en nombre de niveaux de hiérarchie, en taille et largeur de chaque niveau et en nombre de bancs mémoire et de ports : les choix s'effectuent par rapport à des fonctions de coûts s'appuyant sur des paramètres architecturaux, algorithmiques et technologiques issus des bibliothèques de composants mémoires et des spécifications. Une fois la hiérarchie mémoire fixée, on peut procéder à la distribution des données structurées dans les différentes mémoires allouées en se basant sur leurs localités temporelles et spatiales. Le but est de favoriser le stockage des structures les plus souvent accédées au plus proche de l'unité de traitement.

Un premier travail a été mené dans le cadre de la thèse de Sébastien Pignolo, en convention CIFRE avec Matra BAE ; en effet, l'entreprise souhaitait développer une méthodologie d'optimisation de la consommation pour les algorithmes de traitement embarqués dans les missiles, comme ceux de calculs de trajectoire ou de changements de repère [Pign01] [Pign02]. Dans le cadre de ce travail, on se place dans le cas où la hiérarchie mémoire est définie, soit parce qu'on cible un processeur, soit parce que l'étape de dimensionnement de la mémoire a

déjà été réalisée. On sait que, quelle que soit la cible, les accès dans la mémoire interne consomment moins que les accès en mémoire externe. On cherche donc à placer les données le plus souvent manipulées en mémoire interne, mémoire cache ou scratch-pad. La première caractéristique de cette méthode est sa granularité. L'objectif est de diminuer la consommation des applications TDSI pour lesquelles les données manipulées sont essentiellement des données structurées accédées dans des boucles à un ou deux indices, éventuellement imbriquées. Le placement des données structurées en mémoire est dynamique. Comme la hiérarchie mémoire est définie, il reste à trouver où placer les données structurées en mémoire interne et pendant combien de temps ? Pour gérer ce double problème, deux représentations formelles sont utilisées: le Graphe de Dépendance d'Expressions (*GDE*) pour la représentation des dépendances temporelles et le Graphe de Dépendance de Structures (*GDS*) pour la représentation des dépendances structurelles ou spatiales.

Ce travail est actuellement poursuivi avec l'introduction d'une stratégie plus globale permettant d'envisager la définition d'une hiérarchie mémoire pour un processeur (RISC / CISC), un DSP ou un ASIC, ceci à partir d'une spécification algorithmique en C (voire VHDL) de l'application [Mart04]. Dans le cas d'un ASIC, réalisé via l'outil *GAUT* par exemple, on se place en amont de la synthèse d'architecture. L'objectif est de pouvoir définir un ensemble hiérarchie/architecture adéquat tout en faisant une distribution des structures dans cet ensemble, le tout devant respecter des contraintes de consommation et de temps d'exécution.

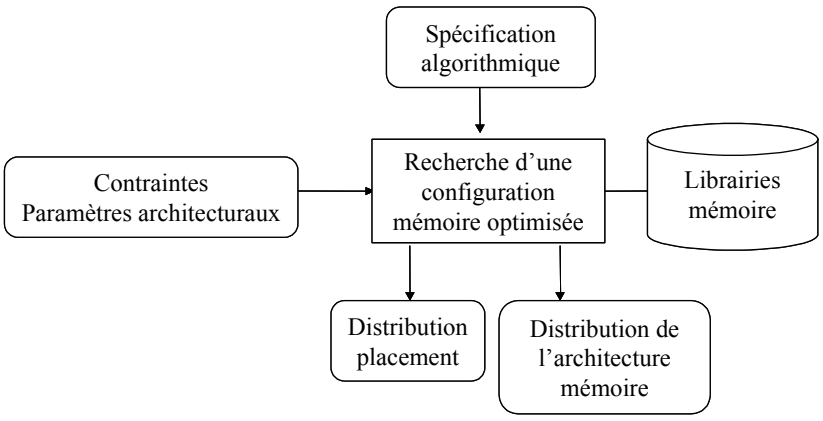


Figure III.2 : Configuration mémoire optimisée.

La liberté d'action est plus grande pour un ASIC ou un FPGA car on peut développer conjointement la partie traitement et la partie relative à la mémoire. De plus il est vraisemblable que la hiérarchie mémoire ne fera pas appel à des systèmes de cache ; s'il est possible de faire une analyse statique du code, alors les transferts de données peuvent être gérés explicitement par une unité spécifique.

Dans le cas des processeurs, la hiérarchie voire l'architecture peuvent être partiellement ou totalement imposées. Pour la synthèse architecturale, le "mapping" mémoire décrivant la distribution et le placement des données structurées peut être défini.

Ce travail est en cours de développement et d'instrumentation ; il permettra à terme de fournir des fichiers de contraintes mémoires qui seront utilisés en entrée de la synthèse d'architecture ou de la compilation.

III.1.3. La synthèse de haut niveau : l'outil GAUT

GAUT (acronyme de Générateur Automatique d'Unité de Traitement) est un environnement de synthèse d'architectures matérielles, dédié aux algorithmes de TDSI sous contrainte de cadence d'itération. A partir de la spécification d'un algorithme en VHDL, il synthétise une description structurelle VHDL de niveau RTL optimisée en surface et destinée aux outils de synthèse logique du marché tels que ISE/Foundation de Xilinx ou Design Compiler de Synopsys. Cet outil universitaire résulte de travaux de recherche commencés au LASTI dans les années 1990 et poursuivis au LESTER depuis 1994 ([Mart93] et [Phil95]). Le développement informatique représente actuellement un volume de l'ordre de 200000 lignes de code C et JAVA.

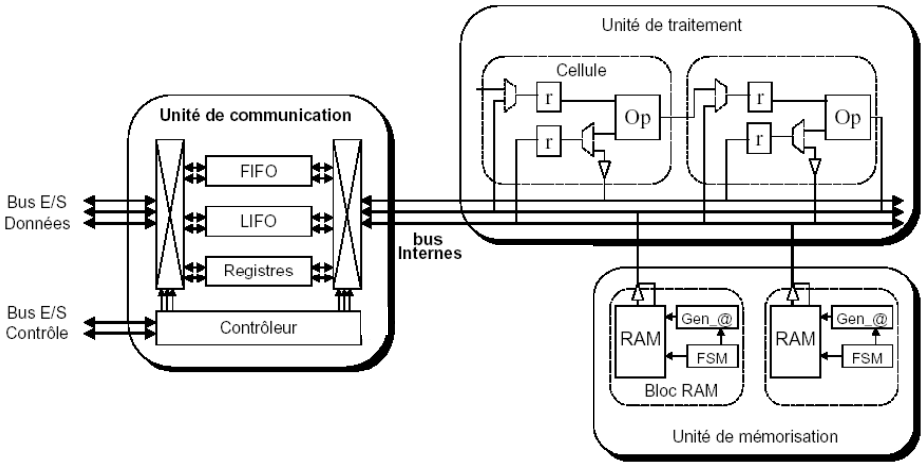


Figure III.3 : Unités fonctionnelles

GAUT est actuellement développé dans le cadre du projet RNRT "ALIPTA" afin de démontrer l'intérêt de la synthèse de haut niveau dans le concept d'IP algorithmique. Cette approche permet la génération de composants virtuels (IP) de niveau RTL à partir d'une spécification de haut niveau, flexible et générique, pour un jeu de contraintes données (cadence, technologie cible, …) [Cass04].

a. Architecture des unités fonctionnelles.

Le modèle cible des architectures synthétisées par *GAUT* est un cœur générique de processeur dédié au traitement du signal *DSP*. Ce modèle est composé de trois unités fonctionnelles travaillant en parallèle : une unité de traitement *(UT),* une unité de mémorisation *(UM),* et une unité de communication *(UCOM)*. Chaque unité dispose d'un chemin de données et d'une ou plusieurs machines à états finis *(FSM)*. Une horloge commune assure le fonctionnement synchrone entre les différentes unités qui fonctionnent en mode pipeline et s'échangent les données au travers d'un réseau multi-bus parallèles.

Unité de mémorisation.

L'unité de mémorisation *(UM)* répond à l'optimisation de stockage des données. Elle est constituée de bancs mémoires *(RAM / ROM)*, de registres et de générateurs d'adresses. Un générateur d'adresse est une machine d'états finis dont chaque état correspond à la lecture de la valeur d'une constante ou à la lecture/écriture de la valeur d'une variable en RAM. La gestion de l'unité de mémorisation sera détaillée dans les chapitres suivants.

Unité de Traitement.

L'unité de traitement *(UT)* est en charge de la partie calcul de l'algorithme. Son modèle s'articule autour de cellules (arithmétiques ou logiques) élémentaires. Chaque cellule est composée d'un opérateur, d'un ensemble de registres, de multiplexeurs, de démultiplexeurs et de buffers trois états. Les registres permettent le stockage temporaire des données et la synchronisation des transferts de données au sein de *l'UT*. Les multiplexeurs, démultiplexeurs et buffers trois états ont en charge l'aiguillage des données. Leur présence optionnelle dans une cellule résulte du partage des registres et des opérateurs, réalisé durant la phase d'optimisation du matériel. Les cellules communiquent au travers d'un réseau de bus parallèles. Les bus dédiés assurent les échanges entre les cellules alors que les bus généraux permettent les accès aux mémoires contenues dans l'unité de mémorisation. En fonction des contraintes temporelles, l'architecture peut être pipeline ou non. Dans le cas d'architectures pipelines, des tranches de pipeline sont rajoutées pour conserver une architecture respectant la cadence d'arrivée des données.

Unité de Communication.

L'unité de communication *(UCOM)* est composée d'une partie opérative *(Chemin de données)* et d'un contrôleur [Baga98]. Le chemin de données inclut des modules de stockage des données d'entrée/sortie *(FIFO, LIFO, registres)*, des bus (*internes et externes*) et des éléments d'interconnexion *(multiplexeurs, démultiplexeurs et barrières trois états)*. Les bus externes

représentent les liens physiques vers les autres composants du système. Les bus internes représentent les liens physiques avec l'unité de traitement. Leur nombre est défini par la synthèse et peut être supérieur ou égal au nombre maximum d'échanges simultanés de données entre l'*UCOM* et l'*UT*. Le contrôleur implémente d'une part le protocole de communication système et d'autre part synchronise l'unité de communication et l'unité de traitement. Dans [Baga97], l'auteur présente le flot de conception de l'*UCOM* à partir des contraintes d'une unité de traitement, synthétisée sous contrainte de cadence, et des contraintes système dans un flot de conception matériel / logiciel. L'auteur démontre que dans certains cas, les contraintes imposées par l'unité de traitement et le système sont trop éloignées pour être adaptées par une interface rendant ainsi la synthèse de l'*UCOM* impossible. Bien qu'elle repose sur une modélisation des échanges entre le composant et un DSP, la méthodologie ne propose pas de gestion explicite des points de synchronisation. Elle utilise de plus une unité de traitement synthétisée sous contrainte de cadence qui ne prend pas en compte les caractéristiques des séquences d'E/S. Les travaux sur la gestion de la communication ont été poursuivis dans la thèse de Philippe Coussy ([Cous03a], [Cous03b]). Dans ces travaux, l'auteur propose une approche de réutilisation des IPs dans les applications orientées traitement du signal, de l'image et des télécommunications. Dans ce contexte, la spécification de l'IP est modélisée par un Graphe Flot de Signaux (SFG) qui, couplé aux temps de propagations des opérateurs et à la cadence d'itération, permet la génération d'un *graphe de contrainte algorithmique* (ACG). Il a développé une analyse formelle des contraintes, qui repose sur les calculs de cycles, et permet de vérifier la cohérence entre la cadence, les dépendances de données de l'algorithme et les contraintes technologiques. Les contraintes d'intégration, spécifiées pour chacun des bus (ports) connectant l'IP aux autres composants du système, sont modélisées par un *graphe de contraintes d'Entrée/Sortie IOCG* (*IO Constraint Graph*) dont la sémantique est issue des travaux de Ku et De Micheli [Ku91]. Ce modèle supporte, entre autre, la modélisation (1) du type de transferts, (2) des variations temporelles des dates d'arrivées des données, (3) du séquencement des données échangées, (4) des mécanismes liés aux protocoles. Les contraintes d'intégration et les contraintes algorithmiques de l'IP sont fusionnées pour fournir un graphe détaillé des contraintes GCG (*Global Constraint Graph*) exhibant les points de synchronisation entre l'environnement et le composant. Des optimisations pour l'implémentation sont proposées à partir de transformations formelles du graphe.

b. Flot de conception.

Le flot de conception mis en œuvre dans *GAUT* peut globalement être représenté par la Figure III.4. La description algorithmique d'une application est spécifiée pour une itération. Elle est décrite dans un sous-ensemble du langage VHDL par un unique processus (couple entité / architecture). La description initiale est accompagnée d'une contrainte temporelle. Cette valeur,

appelée *Tcadence*, correspond à l'intervalle de temps séparant le début des transferts des données d'entrée nécessaires à une itération de l'algorithme et le début des transferts nécessaires à l'itération suivante. La phase de compilation effectue une analyse syntaxique, un contrôle sémantique et une parallélisation du code. Cette étape réalise (1) la suppression des dépendances de contrôle (déroulage des boucles, mise en ligne des appels de procédures, parallélisation des branchements conditionnels) et (2) la suppression des fausses dépendances de données. En effet une distinction entre les vraies dépendances (producteur-consommateur) et les fausses dépendances que sont l'anti-dépendance (consommateur-producteur) et la dépendance de sortie (producteur-producteur) est faite dans l'outil *GAUT*. Le renommage des variables permet l'assignation unique et facilite la parallélisation automatique du code. La compilation produit une représentation interne de l'algorithme sous la forme d'un graphe flot de signaux *(SFG)*. Cette modélisation permet l'expression du parallélisme maximal de l'algorithme. Après l'étape de synthèse de l'unité de traitement, l'unité de mémorisation est générée. Le flot de conception se termine par la synthèse de l'unité de communication qui est réalisée en tenant compte des contraintes d'intégration et des contraintes fournies par la synthèse de l'unité de traitement. Afin de garantir la compatibilité avec le plus grand nombre d'outils de synthèse RTL du commerce, le VHDL RTL produit par *GAUT* est strictement conforme à la norme *IEEE 1076* (*Standard for VHDL Register Transfer Level Synthesis*).

Actuellement le projet RNTL "*SystemCmantic*" étudie la possibilité d'exprimer les spécifications d'entrée de *GAUT* à l'aide du langage SystemC.

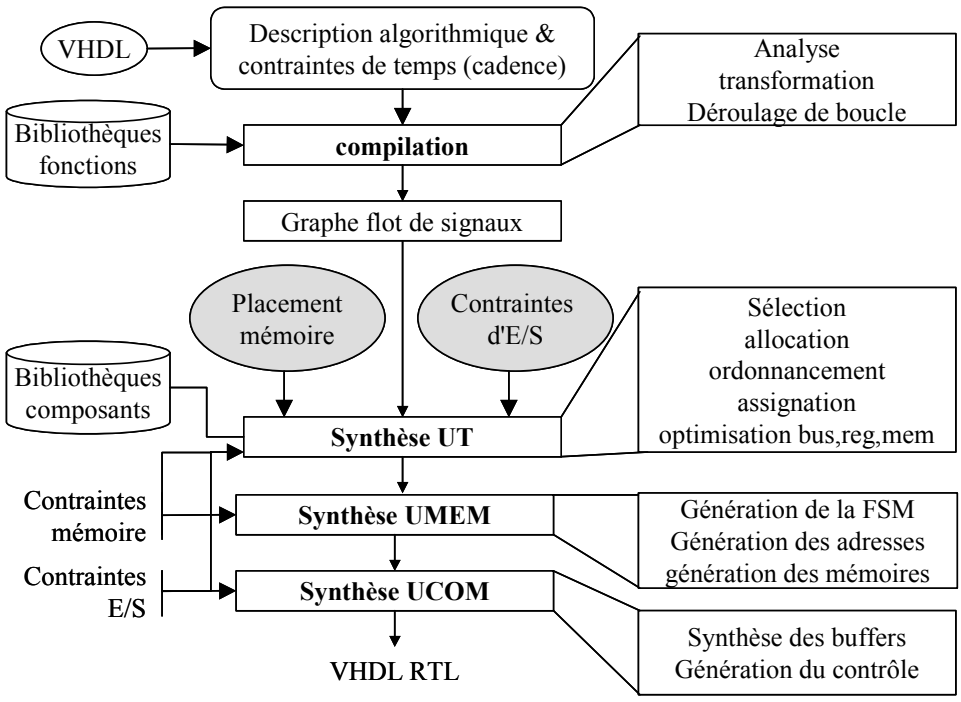


Figure III.4 : Flot de conception de l'outil *GAUT*

III.2. Gestion des unités de mémorisation

III.2.1. Contexte

La spécification d'entrée est une description algorithmique représentant le comportement de l'application sur une itération. L'étape de compilation met en œuvre le mécanisme d'assignation unique pour toutes les données spécifiées dans la description comportementale. Lorsque plusieurs écritures d'une donnée sont exprimées dans la spécification comportementale, la donnée est renommée à chaque nouvelle écriture de manière à respecter l'assignation unique dans la description intermédiaire. L'implantation de l'algorithme peut être effectuée sur une architecture pipeline ou non pipeline. Lorsque la latence de l'architecture est supérieure à la cadence, l'architecture est décomposée en tranches qui correspondent aux différents étages de pipeline de l'architecture.

Les données à mémoriser peuvent être distribuées en registres dans l'unité de traitement ou dans les bancs mémoire de l'unité de mémorisation. La distribution des données en registres ou en mémoire s'effectue en fonction d'un seuil de durée de vie des données au-dessus duquel elles ne peuvent rester en registre interne.

a. Définition de la durée de vie

Parmi l'ensemble des types de données, *GAUT* distingue deux groupes, les données locales et les données globales. La notion de localité est associée à chaque donnée par une évaluation de l'intérêt de sa mémorisation en registre ou en mémoire. Une donnée utilisée fréquemment et durant des laps de temps courts sera donc placée en registre. Cet intérêt est calculé par rapport au type de la donnée, à sa durée de vie et à son nombre d'accès. La durée de vie normalisée *dvn(di)* d'une donnée *i* est calculée par la relation :

$$dvn(d_i)=\frac{dv(di)}{\sum accès(di)}$$

où *dv(di)* la durée de vie de la donnée *i* correspond au temps entre le premier accès et le dernier accès à la donnée et où $\sum accès(di)$ correspond au nombre d'accès à la donnée *i*. Il s'agit de pondérer la durée de vie des données par leur nombre d'accès (lectures et écritures) afin de prendre en considération le taux de réutilisation d'une donnée.

La difficulté consiste à trouver la valeur optimale du seuil de localité. Un seuil trop élevé entraîne la mise en registre d'une grande partie, voire de toutes les variables. Le coût de l'unité de traitement va être élevé puisque les registres sont très coûteux en termes de surface et de consommation. Dans le cas contraire, toutes les données sont placées en mémoire, cela réduit fortement le nombre de registres de l'unité de traitement mais est cependant pénalisant puisque

le nombre d'accès mémoire va être important ce qui risque de diminuer les performances du système.

Un seuil minimum est introduit et est défini par :

$$Seuil_{\min} = 2\Delta mem + \max\left(Ttrav\left(Op_i\right)\right)$$

où Δmem représente le temps d'un accès en lecture ou en écriture à la mémoire, par les composants mémoire définis dans les bibliothèques de composants. $Ttrav(Op_i)$ est le temps de traversée de l'opérateur i. Le seuil de localité n'est pas borné, il peut donc prendre une valeur infinie ce qui correspond à placer toutes les variables en registres sauf les données circulant sur les ports IN/OUT.

b. Propriétés des données à mémoriser

Toutes les données d'un algorithme fourni à l'outil *GAUT* sont représentées par des scalaires. Quatre types de données sont associés aux scalaires d'un algorithme : les constantes, les variables, les données récursives ou vieillissantes.

- Une donnée de type constante est une donnée qui n'est jamais écrite dans la spécification algorithmique.
- Une donnée de type variable est une donnée qui est écrite puis lue dans la même itération de l'algorithme. Une variable peut être exprimée de manière explicite dans l'algorithme ou de manière implicite. Exemple : x := a+b*c ; les variables a, b et c sont des variables explicites. Le résultat de la multiplication b*c doit être conservé de manière à réaliser l'addition du résultat de la multiplication avec la variable a. Le résultat de la multiplication entraîne la création d'une variable dite implicite puisque n'apparaissant pas dans l'algorithme (un nom est associé à chaque variable implicite). La création de cette variable est nécessaire pour respecter la sémantique du graphe flot de signaux (cf définition du graphe flot de signaux chapitre IV).

$$x := a + b * c \text{ revient à : } \begin{array}{l} v\_impl := b * c; \\ x := a + v\_impl, \end{array}$$

- Une donnée récursive est une donnée qui est lue puis écrite à une itération donnée de l'algorithme, elle sera lue à l'itération suivante.
- Une donnée vieillissante est une donnée introduisant un retard entre deux itérations de l'algorithme, opération z^{-1} dans le domaine TDSI. La valeur d'une donnée x_n prend la valeur de la donnée x_{n-1} de l'itération précédente. On définit une fenêtre et un glissement de cette fenêtre sur un vecteur de taille infinie.

c. Sémantique

Nous allons expliciter la sémantique graphique utilisée pour illustrer la gestion des données manipulées dans les applications TDSI. Nous définirons la gestion de ces différents types de données. Puis, à partir d'exemples de pseudo-code, nous représenterons un chronogramme des accès à la mémoire une fois l'ordonnancement effectué. Nous introduirons également les implications d'une implantation pipeline pour chaque type de données et les solutions de distribution et placement qui en découlent.

Les données apparaissent dans la spécification algorithmique comme le montre la Figure III.5 ci-dessous.

```
Exp 1        x:=a+b;
  ⋮
Exp 2        y := x + 2;
```

Figure III.5 : Spécification algorithmique

Nous représentons les accès à la mémoire de données exprimées dans la spécification dans un chronogramme ; les ronds blancs représentent les lectures en mémoire, les ronds gris les écritures. Les accès représentés sont le résultat de l'ordonnancement réalisé. La cadence et la latence sont également exprimées de manière à faire apparaître l'architecture à mettre en œuvre (pipeline ou non pipeline). La Figure III.6 montre les accès mémoire à la donnée *x* de la spécification de la Figure III.5.

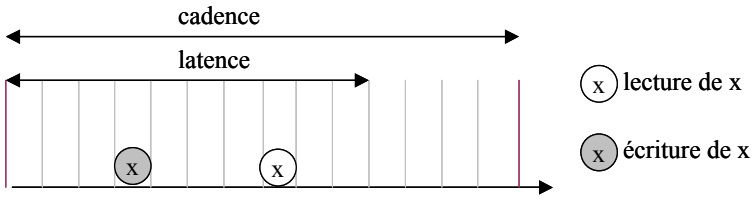


Figure III.6 : Chronogramme des accès mémoire

Lorsque la cadence est inférieure à la latence, l'architecture de l'unité de traitement devient pipeline et des précautions doivent être prises pour gérer la cohérence des données entre les différentes itérations de l'algorithme. Pour chaque type de données, un premier chronogramme illustrera des accès pour une architecture non pipeline puis un second illustrera le mode pipeline. La distribution et le placement des données seront finalement introduits pour les différents types de données.

III.2.2. Gestion des données

Pour chaque type de donnée, il faut définir une stratégie de mémorisation pour gérer le mode pipeline en illustrant le mécanisme de gestion des lectures et/ou écriture.

a. Gestion des constantes

Pour chaque itération de l'algorithme les données de type constante sont seulement lues. La mémorisation des constantes s'effectue en mémoire ROM, les constantes ne sont pas câblées. La Figure III.7 montre deux spécifications utilisant les constantes C1 et C2. La donnée x est un échantillon d'entrée.

Exemple 1	Exemple 2

CONSTANT C1, C2 : *entier*;
Début
a := C1 + x;
b := C2 + x;
c := C1 + b;
d := C1 + a;
e := C2 + c;
f := C1 + d;
Fin
	CONSTANT C1, C2 : *entier*;
	Début
	a := C1 + x;
	b := C2 + x;
	c := C1 + b;
	d := C2 + a;
	Fin

Figure III.7 : Spécifications avec constantes

Les accès aux constantes après ordonnancement sont présentés Figure III.8. Lorsque la cadence est supérieure à la latence, il suffit pour obtenir une architecture mémoire cohérente de garantir qu'il n'y ait pas de conflits d'accès sur une itération. Dans les deux exemples, il n'y ait pas d'accès simultanés à la mémoire, donc les constantes C1 et C2 peuvent être distribuées dans la même mémoire.

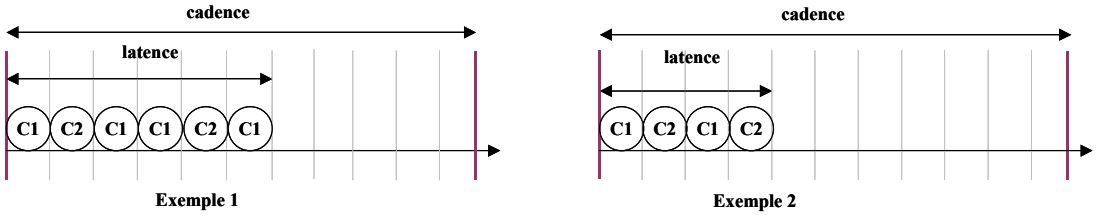


Figure III.8 : Chronogramme des accès aux constantes (cadence > latence)

Lorsque la latence est supérieure à la cadence, nous obtenons une architecture pipeline pour laquelle des conflits d'accès à la mémoire peuvent se produire. Pour éviter ces conflits d'accès aux constantes, il suffit d'assigner une mémoire et une adresse à toutes les constantes et de vérifier qu'il n'existe pas de conflits entre les différentes tranches de l'architecture. Si des conflits apparaissent, il faut revoir l'architecture mémoire.

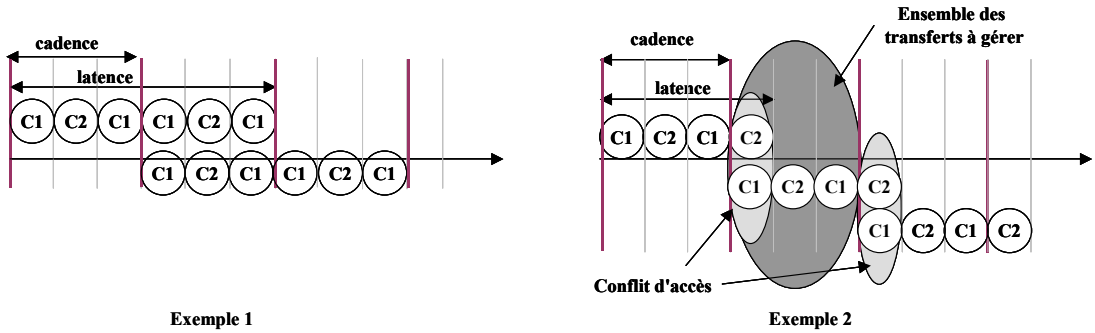


Figure III.9 : Chronogramme d'accès aux constantes (cadence < latence)

Dans l'exemple 1 de la Figure III.9, les accès aux constantes au temps modulo la cadence se font sur des données identiques, il n'y a donc pas de conflit d'accès. Dans l'exemple 2, les constantes C1 et C2 sont accédées en parallèle, il y a un conflit qui peut être évité en plaçant les constantes C1 et C2 dans deux bancs mémoire distincts ou en utilisant une mémoire double ports. La gestion des transferts devra se faire en prenant en compte tous les transferts des tranches de pipeline.

b. Gestion des variables

Pour ces données qui sont écrites puis lues lors d'une même itération de l'algorithme, il faut aussi gérer les conflits d'accès à la mémoire au sein des différentes tranches de l'architecture pipeline. L'assignation unique est réalisée pour les variables écrites puis lues plusieurs fois dans une itération de l'algorithme. Une variable *a* peut ainsi prendre plusieurs valeurs au cours de l'itération ; elle sera renommée autant de fois que nécessaire pour permettre de différencier ses valeurs successives.

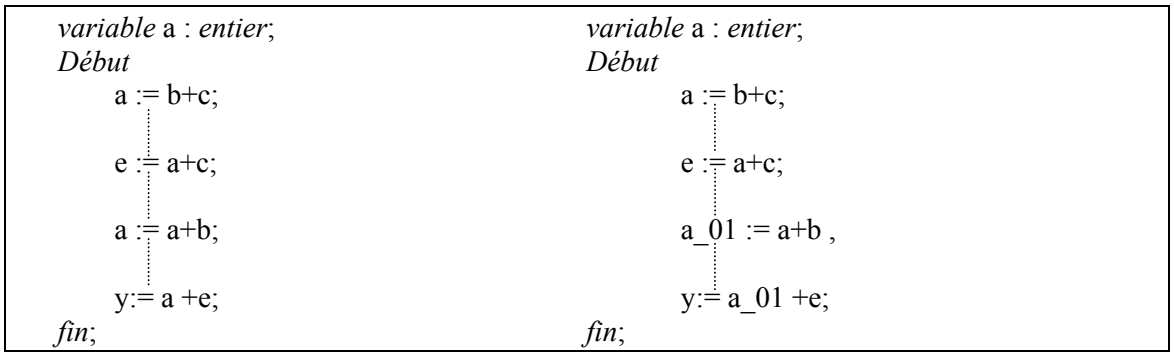


Figure III.10 : Spécification de variables

Dans cet exemple, la donnée *a* est du type variable puisqu'elle est écrite, lue, écrite puis à nouveau lue sur une même itération. L'assignation unique revient à renommer la donnée *a* lors de sa deuxième écriture comme le montre la spécification de droite de la Figure III.10.

Les données *a* et *a_01* sont deux variables différentes, cependant elles peuvent être placées en mémoire à la même adresse physique s'il n'y a pas recouvrement de leurs durées de vie respectives.

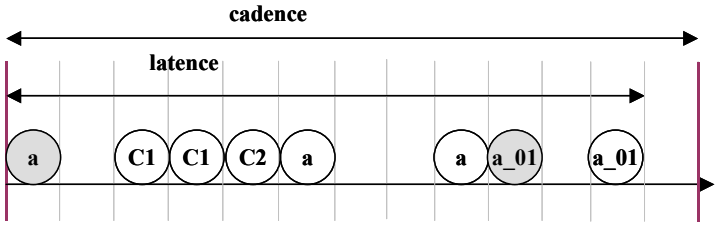


Figure III.11 : Architecture non pipeline

Lorsque la contrainte de cadence devient plus forte, l'architecture de l'unité de traitement devient une architecture pipeline. Il faut alors garantir qu'une écriture d'une donnée à l'itération *i* n'écrase pas la valeur de la donnée de l'itération *i-1* qui doit encore être lue. La Figure III.12 illustre ce phénomène, le cycle grisé indique qu'il y a une écriture d'une nouvelle valeur de *a* avant la dernière lecture de l'itération précédente. Le calcul ne sera pas, dans ce cas, effectué avec la bonne valeur de *a*. Dans ce cas, la donnée *a* de l'itération *i+1* sera lue dans l'itération *i*. Il faut donc mettre en place un mécanisme de conservation de données ; chaque donnée dont la date modulo la cadence de la dernière lecture est supérieure à la date modulo cadence de l'écriture devra être dupliquée en mémoire. Pour notre exemple, deux emplacements mémoire devront être réservés pour stocker les données *a* de l'itération *i* et *i+1*. Une gestion simple par FIFO peut être envisagée, elle permet de garantir la cohérence des données pour des architectures pipelinées.

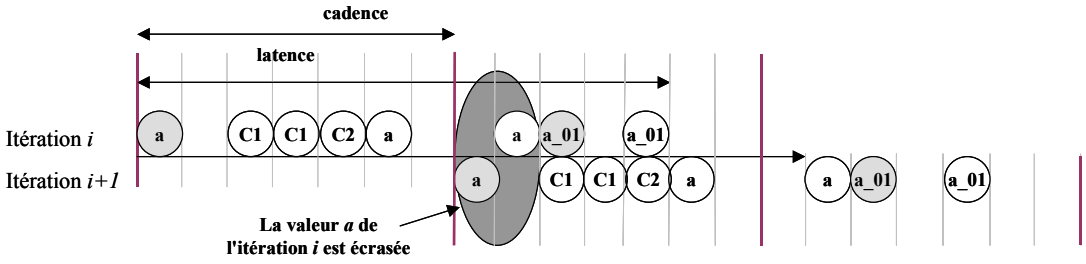


Figure III.12 : Gestion des variables pour les architectures pipeline

c. Gestion des données récursives

Les données récursives sont lues dans l'itération courante de la spécification puis écrites durant l'itération suivante. La valeur de la donnée écrite à l'itération courante doit être conservée pour la lecture de l'itération suivante. Les données récursives peuvent être gérées comme les données de type variable lorsque les architectures générées ne sont pas pipeline.

```
variable a,b,c,e : entier;
Début
        a := b+4;
        b := x1+b;
        e := a+ c;
        c:= x2 +e;
fin;
```

Figure III.13 : Spécification de données récursives

Dans la Figure III.15, les données *b* et *c* sont des données récursives puisqu'elles sont d'abord lues puis ensuite écrites. Les nouvelles valeurs de *b* et *c* calculées servent à l'itération suivante. Le chronogramme des accès aux données récursives pour une architecture non pipeline montre que, comme pour les variables, il n'y a pas de problème de gestion des données.

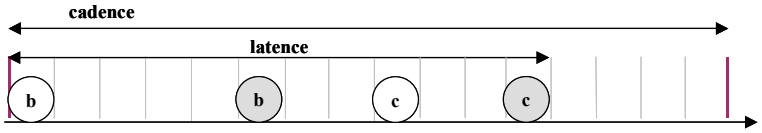


Figure III.14 : chronogramme d'accès de données récursives

Un problème d'ordonnancement peut se produire lors de l'apparition de tranches de pipeline dans l'architecture. La Figure III.15 représente les chronogrammes des accès aux données récursives pour deux ordonnancements différents. Dans l'exemple 1, le temps entre la première lecture et l'écriture d'une donnée est toujours inférieur à la cadence. Dans ce cas, il n'y a pas de problème d'ordonnancement. Dans l'exemple 2, la durée entre la lecture et l'écriture de la donnée *b* est supérieure au temps de cadence. La donnée *b* lue à l'itération *i+1* ne sera pas la donnée *b* produite à l'itération *i* mais celle produite à l'itération *i-1*.

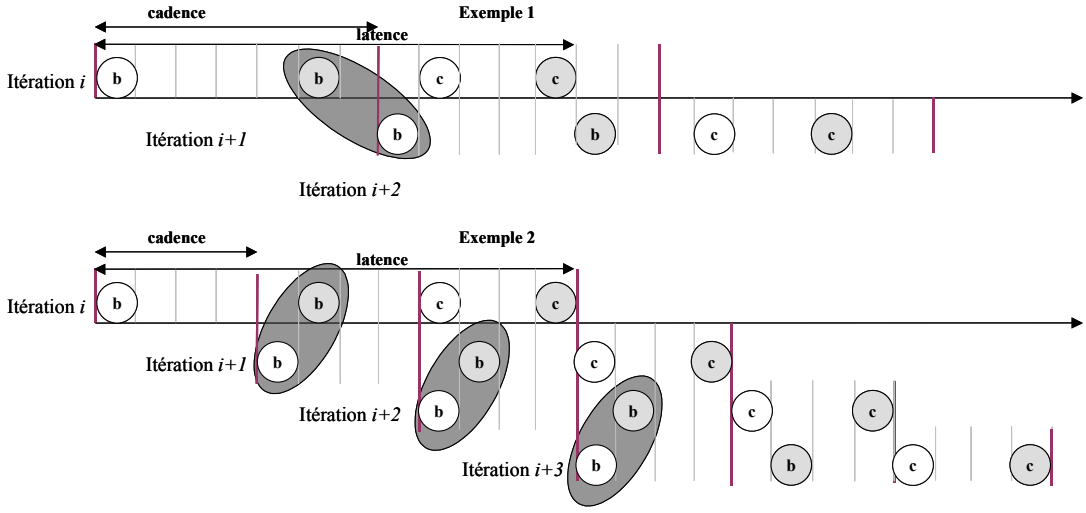


Figure III.15 : Chronogrammes d'accès aux données récursives pour des architectures pipeline

Il n'y a pas de solution de synthèse pour l'exemple 2 ; la contrainte de cadence doit être relâchée pour obtenir une solution. Il faut écrire les données récursives au plus tôt de manière à garantir que les données servant aux calculs de l'itération suivante soient disponibles le plus tôt possible.

d. Gestion des données vieillissantes

Les données vieillissantes devront être gérées de manière à prendre en compte le retard introduit entre deux variables d'une itération à l'autre de l'algorithme. Ce retard est décrit de manière explicite dans la spécification algorithmique.

Dans la spécification suivante (Figure III.16), un vecteur X est défini et les données du vecteur évoluent en fonction du glissement d'une fenêtre temporelle. Le détail du mécanisme de vieillissement est présenté dans le chapitre suivant. Les données sont lues pour effectuer les opérations de l'unité de traitement. Le vieillissement, explicite dans la spécification, permet l'obtention des valeurs correspondant à la fenêtre de l'itération suivante.

```
Variable x : vecteur
Début;
x(0):=x_inpout    :
tmp := x(0);
for (i=1 ; i =N-1; 1++)  -- traitement utilisant les données vieillissantes
      tmp = tmp+ x(i);
      :
For (i=N-1 ; i= 1 ;i--)  -- vieillissement explicite des données du vecteur x
      x(i)=x(i-1)
fin;
```

Figure III.16 : Spécification de données vieillissantes

Pour une architecture non pipeline, il suffit de s'assurer que les données du vecteur vieillissant ont évoluées d'une itération à l'autre. L'évolution se fait soit par une réécriture de toutes les données du vecteur, soit par un décalage de l'adresse correspondant au glissement de la fenêtre (cf chapitre suivant).

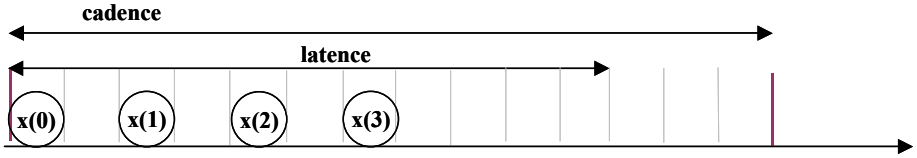


Figure III.17 : Chronogramme d'accès du vecteur vieillissant

L'introduction de pipeline entraîne la plupart du temps des conflits d'accès à la mémoire. Par exemple, pour un algorithme contenant des données vieillissantes tel qu'un filtre FIR, tous les échantillons vieillissants peuvent être contenus dans un seul et unique banc mémoire si l'itération de l'algorithme est contenue dans la cadence. Lorsque la cadence diminue, il y a

apparition d'une architecture pipeline et les conflits d'accès ne peuvent être résolus sans introduire de nouveaux bancs mémoire.

Dans l'exemple de la Figure III.18, l'échantillon x(2) du vecteur x de l'itération i+1 doit avoir pris la valeur de l'échantillon x(1) de l'itération i+1 dans un temps inférieur au temps de cadence.

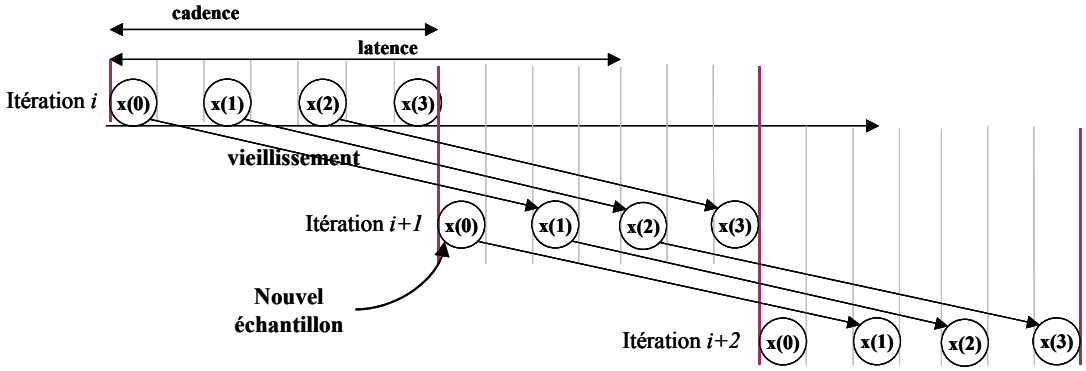


Figure III.18 : Chronogramme d'accès du vecteur vieillissant pour une architecture pipeline

Un mécanisme de distribution des données sur plusieurs bancs mémoire peut être mis en œuvre en fonction du nombre de tranches, du nombre d'accès par tranche, des séquences d'accès aux données d'un vecteur et de la fenêtre de vieillissement. La gestion des données vieillissantes sera détaillée dans le chapitre suivant où un formalisme et une approche dédiés à ce type de données seront introduits.

III.2.3. La synthèse d'architectures sous contraintes mémoires

La synthèse mémoire consiste à intégrer les contraintes liées à la distribution en mémoire durant les phases classiques de synthèse architecturale de l'unité de traitement. La distribution définie en amont sera intégrée comme une nouvelle contrainte de synthèse en entrée de l'outil *GAUT*. Cette nouvelle contrainte, ainsi que les propriétés des différents types de données, seront prises en compte lors des différentes étapes de la synthèse et principalement lors de l'ordonnancement des opérations. L'ordonnancement de l'application doit en effet tenir compte d'éventuels conflits d'accès aux données en fonction de leur distribution dans la hiérarchie et des types de mémoires sélectionnés. Ce nouveau modèle d'ordonnancement sous contraintes mémoire va permettre de réaliser un compromis entre les optimisations de l'unité de traitement et de l'unité de mémorisation.

La dernière étape de la synthèse de haut niveau consiste à définir les générateurs d'adresses pour chaque niveau de hiérarchie. Pour cela, il faut s'intéresser au placement des données scalaires que la synthèse d'architecture a défini comme étant des données à placer en mémoire. Ces données sont des données temporaires, engendrées par le traitement, qui ont une durée de

vie suffisamment importante pour ne pas être stockées en registre dans l'unité de traitement. Il faut donc effectuer le placement des données dans les bancs mémoire, ce qui revient à affecter une adresse physique à chaque donnée. Cette étape est fortement liée à la construction des générateurs d'adresses, qui peuvent utiliser des techniques d'encodage afin de diminuer le nombre de transitions sur les bus d'adresses.

La réalisation d'un ordonnancement et la synthèse des générateurs d'adresses doivent être fiables et automatiques. Pour cela nous introduirons une approche formelle à base de graphes conduisant à un ordonnancement et une synthèse sûrs.

III.3. Conclusion

Notre approche, en intégrant la mémorisation dans la synthèse d'architecture, doit permettre de prendre en compte les différents types de données et de proposer des solutions de stockage pour chacune d'entre elles. Elle intègre :

- la spécification des données utilisées en TDSI et leurs décompositions en classes
- le placement et l'allocation des données structurées ; la distribution des données en mémoire permet de définir de nouvelles contraintes d'accès mémoire pour des architectures parallèles et/ou pipeline
- l'ordonnancement des opérations des unités de traitement sous contraintes mémoire
- la synthèse des générateurs d'adresses

Cette stratégie devra augmenter les possibilités d'optimisation de l'unité de mémorisation car elles ne seront plus contraintes par l'architecture de l'unité de traitement.

La prise en compte des contraintes de mémorisation lors de la synthèse d'architecture repose sur des modèles formels dont la projection va permettre la définition d'architectures d'unités de mémorisation efficaces.

Chapitre IV
Approche formelle

La synthèse doit intégrer la conception de l'unité de mémorisation. Nous proposons une approche formelle qui permettra d'intégrer les contraintes de mémorisation dans la synthèse de haut niveau. Une première approche propose une modélisation des contraintes mémoire. Ces modèles seront ensuite projetés sur une architecture mémoire. Cette approche est ensuite complétée en intégrant la gestion de vecteurs vieillissants pour des architectures mémoire multi-bancs.

IV.1. Synthèse de l'unité de traitement

Comme nous l'avons mentionné dans le chapitre III, l'architecture ciblée par le flot de synthèse de *GAUT* est composée de trois unités fonctionnelles : l'unité mémoire, l'unité de traitement et l'unité de communication. Dans cette première partie, nous détaillons le flot de synthèse, sous contrainte de cadence d'itération, de l'unité de traitement. Dans l'environnement *GAUT*, cinq grandeurs de temps sont utilisées pour la synthèse de l'unité de traitement :

- *Tfonc* : le temps de fonctionnement des opérateurs défini en bibliothèque
- *Phase_min* : le temps de cycle de l'horloge fixé par l'utilisateur. La période de l'horloge doit être supérieure au plus petit temps de traversée des opérateurs
- $Tcycle = \mathrm{Max}\{Tfonc\}_{opérateurs\ sélectionnés}$: le plus grand temps de traversée d'un opérateur
- *Tcadence* : le temps de cadence pour l'exécution de l'application
- *Tlatence* : le temps d'exécution de l'algorithme (chemin critique) qui, s'il est supérieur à *Tcadence*, impose la mise en œuvre d'une architecture de traitement pipeline

IV.1.1. Sélection / Allocation

La synthèse de l'unité de traitement débute par la sélection des opérateurs. Cette phase permet d'associer des délais *Tfonc* aux opérations. Ces délais sont, au préalable, ajustés pour être des multiples entiers de la période d'horloge *phase_min*. La mobilité des opérations est ensuite calculée en utilisant les dates au plus tôt *(ASAP)* et au plus tard *(ALAP)* [Gajs91] de chaque nœud du graphe. Pour cela, les données d'entrée ont une date au plus tôt fixée à t=0, et les sorties une date au plus tard fixée à *Tlatence*. L'allocation dans *GAUT* consiste à fixer un nombres d'opérateurs pour chaque tranche de pipeline par rapport au nombre d'opérations à réalisées par tranche.

La complexité de la phase d'allocation est en *O(n)*, où n représente le nombre d'opérations à traiter. Le nombre de tranches *Tranche* de pipeline est défini par :

$$Tranche = Tlatence / Tcadence$$

où *Tlatence* représente la date maximum au plus tôt (ASAP) des sorties.

Afin de connaître la quantité d'opérateurs à mettre en œuvre pour satisfaire le parallélisme moyen de l'application, le nombre moyen d'opérateurs par tranche est calculé comme suit :

$$Nb\_opr(f,t) = \frac{Nb\_ops(f,t) * tps\_fonct(f)}{Tcadence} \quad \forall \begin{matrix} 0 < t < tranche - 1 \\ 1 < f < Nb\_fonc\_Max \end{matrix}$$

où *Nb_ops(f,t)* est le nombre d'opérateur de type *f* alloués à la tranche *t*, *tps_fonct(f)* le temps de traversée des opérateurs réalisant la fonction *f* et *Tcadence* la contrainte de cadence fixée par l'utilisateur.

Si l'allocation basée sur le nombre moyen d'opérations est respectée durant l'ordonnancement alors la meilleure solution en terme de surface est obtenue. Cependant si le nombre d'opérateurs alloués est insuffisant, on observe une élongation du temps qui entraîne l'ajout de tranches de pipeline et la création de nouvelles ressources.

La technique d'allocation, basée sur le nombre moyen d'opérateurs est adaptée aux synthèses contraintes par les ressources matérielles.

IV.1.2. Ordonnancement / Assignation

L'objectif est de réduire le temps nécessaire à l'ordonnancement des opérations pour obtenir rapidement une solution. La solution consiste à utiliser un algorithme d'ordonnancement ayant une faible complexité. Dans ce contexte nous utilisons un algorithme de type glouton : une décision n'est prise qu'en fonction des événements passés et ne peut être remise en cause par la suite. Un des objectifs de l'ordonnancement implanté dans *GAUT* est de réguler le parallélisme du Graphe Flot de Signaux au parallélisme moyen de l'architecture (cf sélection) tout en limitant l'élongation du graphe. En effet plus l'allongement est important plus le nombre de tranches et donc le nombre d'opérateurs est grand. L'algorithme utilisé, de type *list scheduling* (voir [Gajs91]), est basé sur la gestion des priorités des opérations à exécuter et permet de maîtriser l'élongation du *SFG*. Un autre objectif de l'ordonnancement est d'optimiser l'utilisation des opérateurs ; pour cela, une priorité d'utilisation est associée aux opérateurs qui sont ainsi réutilisés dès leur libération. La progression du temps se fait donc par événement et non de manière discrète. Elle se fait uniquement sur les événements de relâchements des opérateurs et en fonction du *Tcycle*. Les données lues sur les ports d'entrées sont supposées être présentes aux dates assignées par l'ordonnancement, ou lorsque la synthèse est réalisée sous contraintes d'E/S, aux dates fixées par le fichier de contraintes. Des renseignements complémentaires sur l'ordonnancement réalisé sont fournis en Annexe A.

La Figure IV-1 décrit l'algorithme d'une tentative d'ordonnancement. Il est important de noter qu'avant de commencer la phase d'assignation, il existe une phase de création éventuelle d'opérateurs n'ayant pas été prévus par l'allocation. En effet, certaines opérations ne peuvent être allouées dans la tranche de pipeline où elles étaient initialement prévues. On observe dans ce cas une élongation du graphe qui nécessite la création d'opérateurs supplémentaires pour traiter ce parallélisme "dur" (1). Par ailleurs, lorsqu'un opérateur est utilisé sur l'ensemble des cycles d'une tranche de pipeline, il ne peut plus être utilisé sur les tranches de pipeline suivantes. Cet opérateur est dit "usé" et est retiré de la liste des opérateurs disponibles (2).

```
Sélection et allocation;
Initialisation de l'ordonnancement;
Tant qu'il reste des opérations à exécuter faire
    Création éventuelle d'opérateur pour traiter le parallélisme dur (1);
    Chercher à assigner toutes les opérations exécutables
    Progression du temps;
    Rechercher les opérations en cours d'exécution terminant ou relachant leurs opérateurs;
    Libérer les opérateurs aux opérations se terminant;
    Déterminer les nouvelles opérations exécutables;
    Si nouvelle tranche;
        Créer les opérateurs prévus par l'allocation;
Détecter les opérations en attente;
Supprimer les opérateurs usés (2);
Fin tant que;
```

Figure IV-1 : Tentative d'ordonnancement

IV.2. Modèles et gestion pour la synthèse sous contraintes mémoire

IV.2.1. Objectifs

A l'instar de la compilation sur les DSP de dernière génération, nous considérons que le concepteur a étudié tout ou partie de la distribution et du placement des structures de données de son algorithme.

Cette tâche est délicate car elle influe directement sur les performances de l'architecture à synthétiser. Nous présenterons à la fin du document comment, dans une approche itérative, le concepteur peut progressivement décider de la distribution en mémoire.

Dans une première approche, le processus de synthèse peut paraître relativement simple. Il suffit lors de l'ordonnancement de l'unité de traitement, de s'assurer que les données calculées sont accessibles en mémoire. C'est à dire :

- on ne peut ordonnancer un traitement que si ses prédécesseurs sont traités **et** que les données sont lues en mémoire ;
- on ne peut lire qu'une donnée par banc mémoire et par cycle d'accès mémoire (le cycle d'accès à la mémoire peut être différent du cycle de l'unité de traitement).

Dans un deuxième temps nous introduirons la fonction de vieillissement d'un signal (ou fenêtre glissante). Cette fonction d'accès souvent implémentée sous la forme d'un buffer circulaire pose une difficulté particulière dans le cadre d'une distribution d'un signal vieillissant sur plusieurs bancs. Nous avons développé pour résoudre ce problème important du TDSI une approche qui repose sur un modèle de graphe et une projection sur un modèle d'architecture générique.

IV.2.2. Description du flot

A partir de la spécification comportementale de l'application, exprimée sous forme d'algorithme, un Graphe Flot de Signaux (*SFG*) est généré. Dans notre approche, ce graphe est analysé pour extraire toutes les données de l'application et pour déterminer une représentation de leurs distributions. Les données peuvent être placées en registres ou en mémoires suivant leur type. Les données distribuées en mémoire vont définir une distribution mémoire qui va contraindre la synthèse de l'unité de traitement et de l'unité de mémorisation. Les caractéristiques d'accès à la mémoire sont prédéfinies dans une bibliothèque matérielle.

IV.2.3. Modèle de représentation pour la synthèse sous contraintes mémoire

a. Graphe Flot de Signaux (SFG)

La spécification d'entrée de la synthèse de haut niveau est une description fonctionnelle ou algorithmique d'une application. Il est possible d'extraire un graphe flot de signaux ou Signal Flow Graph (*SFG*) de cette spécification algorithmique (Figure IV-2). Le graphe flot de signaux fait apparaître le parallélisme à grain fin des traitements (qui feront l'objet d'un ordonnancement lors de la synthèse), et le retard "applicatif" (opérateur Z^{-1} en automatique et TDSI) qui porte en particulier sur le vieillissement des signaux et les signaux récursifs de la spécification.

Définition : un graphe flot de signaux est un graphe polaire orienté *GFS(V, E)* où l'ensemble des nœuds $V = \{v_0, ..., v_n\}$ représente les opérations, v_0 et v_n étant respectivement le nœud source (déb) et le nœud puit (fin). L'ensemble d'arcs $E= \{(v_i, v_j)\}$ représente les dépendances entre les nœuds opérations.

Le graphe flot de signaux contient $|V| = n+1$ nœuds. Un nœud représente soit une des opérations suivantes (opération arithmétique, logique ou délai), soit une donnée. Un nœud opération sera toujours précédé par un nœud de donnée. La différence entre un graphe flot de signaux et un graphe flot de données provient des opérations de type *délai.* Ces opérations sont utilisées dans le domaine du traitement du signal pour exprimer l'utilisation d'une donnée dont la valeur est calculée dans une itération précédente de l'algorithme. Un arc $e_{ij} = (v_i, v_j)$ représente une dépendance de données $v_i \succ v_j$ entre les opérations v_i et v_j telle que pour toute itération de *GFS*, l'opération v_i doit démarrer son exécution avant celle de v_j. Pour les dépendances de données, l'exécution de v_j ne peut commencer qu'après la complétion de l'opération v_i.

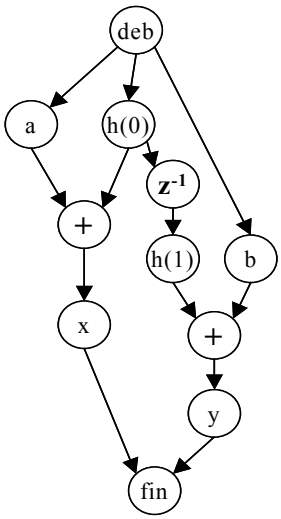


Figure IV-2 : Graphes flot de signaux

b. Graphe de contraintes mémoire

Objectif : pour chaque mémoire, il s'agit de représenter les contraintes fonctionnelles d'accès à la mémoire à partir d'une spécification algorithmique et de ses caractéristiques technologiques.

Pour cela, nous construisons le graphe de contraintes mémoire. Il représente les conflits d'accès aux données et exprime les possibilités d'ordonnancement des opérations dont les opérandes sont placés en mémoire. Un Graphe de Contraintes Mémoire (*MCG* : Memory Constraint Graph) est construit à partir du *SFG*, de la distribution de données en mémoire, et des contraintes technologiques de la mémoire. Il est utilisé durant la phase d'ordonnancement de la synthèse d'architecture

> **Définition** : un graphe de contraintes mémoire est un graphe cyclique orienté, polaire et pondéré *MCG(V';E';W')* où $V' = \{v'_0.:.:; v'_n\}$ est l'ensemble des nœuds de données placées en mémoire. Un graphe de contraintes mémoire contient $|V'| = n + 1$ nœuds qui représentent la taille mémoire en terme de nombre d'éléments à mémoriser. L'ensemble des arcs $E' = \{(v'i; v'j)\}$ représente la précédence entre deux nœuds mémoire. Les arcs sont pondérés par un poids *W'* qui représente le temps nécessaire pour réaliser un accès suivant le placement des données et les contraintes technologiques des mémoires.

Nous allons nous intéresser aux principales contraintes technologiques qui doivent être prises en compte dans la représentation du graphe de conflits mémoire. Nous allons introduire une représentation pour

- les accès aléatoires des mémoires SRAM.
- les accès aléatoires et séquentiels des mémoires DRAM.
- les accès parallèles des mémoires multi-ports.

Chacune de ces représentations sera illustrée par les graphes de contraintes mémoire d'un filtre lms 4 points. Les échantillons *x(i)* sont placés consécutivement dans un banc mémoire. Les coefficients *h(i)* sont placés consécutivement dans un second banc mémoire.

Les mémoires SRAM : les accès aléatoires sont représentés par les arcs $E_0 = \{(v_{0i}; v_{0j})\}$ pondérés par *W'* qui représente dans ce cas le temps d'un accès aléatoire (arcs pleins dans la Figure IV-3 ci-dessous). Ici le *MCG* est constitué de 2 cliques, correspondant chacune à un banc mémoire.

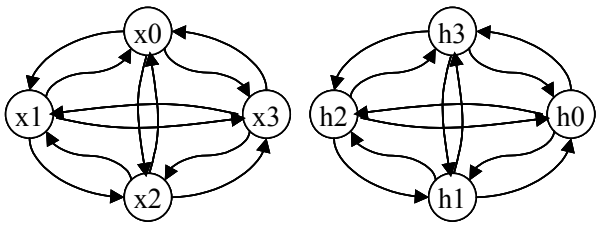


Figure IV-3 : MCG, mémoire SRAM simple port

Les mémoires DRAM : les mémoires DRAM offrent deux possibilités d'accès, des accès séquentiels et des accès aléatoires. Pour le modéliser dans le graphe, *W'* prend deux valeurs distinctes *Wseq* (accès séquentiel) et *Walea* (accès aléatoire). La valeur du poids entre deux arcs dépend du placement des données en mémoire. Si seule la distribution des données a été réalisée, la représentation des accès se limitera à des accès aléatoires.

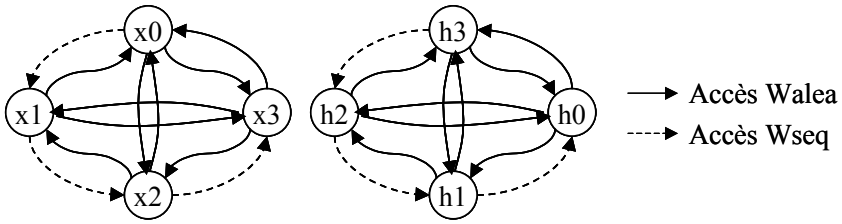


Figure IV-4 : MCG, mémoire DRAM simple port

La Figure IV-4 montre les graphes de contraintes mémoire pour des mémoires DRAM simple port. Les arcs pointillés représentent les accès séquentiels et les arcs pleins représentent les accès aléatoires. La séquence d'accès la plus courte du graphe de contraintes mémoire devra être privilégiée lors de l'ordonnancement des opérations de l'unité de traitement de manière à garantir la vitesse la plus grande de façon à garantir un étirement minimum du SFG.

Les mémoires multi-ports : l'introduction de mémoires multi-ports peut être modélisée par autant de jetons sur les nœuds du *MCG* qu'il y a de ports mémoire. Soit *P* le nombre de ports d'une mémoire, on introduit *P* jetons dans le graphe de contraintes mémoire, chaque jeton *P(i)* représente un accès mémoire à travers le port *P(i)*. La Figure IV-5 modélise les contraintes mémoire pour des mémoires SRAM double ports.

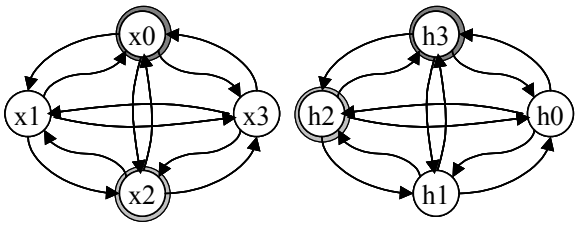


Figure IV-5 : MCG, SRAM double ports

IV.2.4. Modèle architectural de l'unité de mémorisation

Les modèles présentés pour gérer les contraintes mémoires dans la synthèse de haut niveau impliquent la présence d'un modèle architectural de la mémoire. Dans ce cadre, le modèle architectural est relativement simple. Il est constitué d'un compteur d'adresses générant les adresses physiques pour chaque mémoire. La Figure IV-6 représente le modèle architectural, à base de SRAM simple port, de l'unité de mémorisation pour la spécification algorithmique du filtre lms.

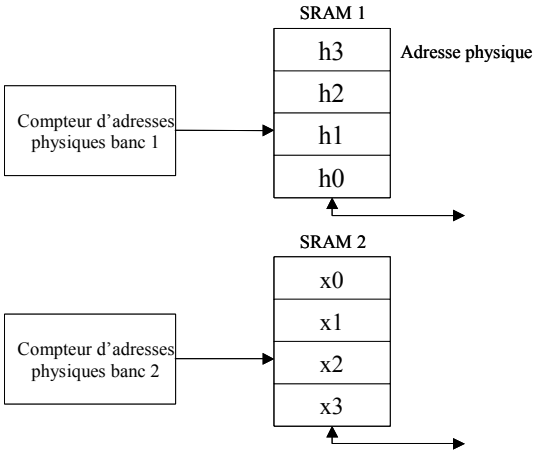


Figure IV-6 : Modèle architectural de l'unité de mémorisation

Ce modèle architectural et les modèles de représentation des contraintes mémoire seront reconsidérés pour la gestion des accès à des vecteurs vieillissants dont les données sont distribuées sur plusieurs bancs mémoire.

IV.2.5. Ordonnancement sous contraintes mémoire

Comme nous l'avons déjà dit, l'ordonnancement des traitements utilisé dans l'outil *GAUT* est un ordonnancement par liste de priorités basé sur une heuristique qui classe les opérations exécutables selon leur mobilité. Lorsque plusieurs opérations ont la même mobilité, elles sont départagées par leur marge. La marge est calculée à chaque nouveau cycle comme étant la date

au plus tard moins le temps courant. Dans le cas d'accès à la mémoire nous introduisons un nouveau critère pour respecter les contraintes mémoire.

Lorsque deux opérations ont leurs opérandes placées dans la même mémoire, l'opération qui a la plus faible mobilité ou, le cas échéant, la plus petite marge est ordonnancée, l'autre opération est retardée. Une opération de l'unité de traitement peut être ordonnancée lorsque les nœuds de donnée du graphe flot de signaux qui doivent accéder à la mémoire respectent le critère d'accessibilité à la mémoire.

> **Définition du critère d'accessibilité à la mémoire** : un nœud de donnée devant accéder à la mémoire est considéré comme accessible si un ou plusieurs ports de la mémoire sont libres au cycle d'accès mémoire courant. Un cycle d'accès à la mémoire représente le temps nécessaire à la réalisation d'une lecture ou d'une écriture pour une mémoire donnée. Ce cycle d'accès peut varier d'une mémoire à l'autre en fonction des mémoires qui ont été sélectionnées.

Toutes les opérations dont les données ont été lues en mémoire ou sont présentes en registre dans l'unité de traitement peuvent être ordonnancées.

Le choix des opérations exécutables en fonction des possibilités d'accès aux mémoires est effectué par un parcours du graphe flot de signaux et un parcours des graphes de contraintes mémoire. La Figure IV-7 présente l'ordonnancement sous contraintes mémoire et les parcours de graphes qui y sont associés. Les lignes en gras représente les modifications qui ont été apportées pour prendre en compte les contraintes d'accès aux mémoires.

Tant qu'il reste des opérations non exécutées *faire*
Parcours du SFG;
Remplir la liste des opérations exécutables;
***Si* date du SFG = date du MCG; (1)**
***Parcours du MCG*;**
Rechercher les données privilégiant les accès séquentiels; (2)
Remplir la liste des données accessibles;
Date du MCG = Date du MCG + cycle du MCG; (3)
Fin *si*
Remplir la liste des opérations exécutables et accessibles; (4)
Ordonnancer les opérations exécutables et accessibles; (5)
Date du SFG = Date du SFG + cycle du SFG;
Fin *faire*

Figure IV-7 : Ordonnancement sous contraintes mémoire

Tant qu'il reste des opérations à ordonnancer, le Graphe Flot de Signaux est parcouru pour trouver les opérations exécutables. Si le temps courant du *SFG* est le même que le temps courant du *MCG* alors, le Graphe de Contraintes Mémoire est parcouru. Parmi les données distribuées en mémoire (opérandes des opérations exécutables) les données à lire en priorité sont recherchées dans chaque clique du *MCG* (2). La priorité est aux données permettant de

séquentialiser des accès à la mémoire. Chaque clique permet de trouver une donnée ou plusieurs données à lire en fonction de la disponibilité ou non des ports de chaque mémoire. Lorsqu'il n'y a plus d'accès possibles, la date du *MCG* évolue en ajoutant un cycle d'accès mémoire au temps courant du *MCG* (3). La liste des opérations exécutables et accessibles contient toutes les opérations dont les prédécesseurs sont faits et dont les opérandes sont en registres ou dans la liste des données accessibles (4). Les opérations contenues dans cette liste peuvent être ordonnancées (5).

IV.3. Modèles et gestion spécifique du vieillissement

IV.3.1. Objectifs

Nous avons identifié des fonctions d'adressage avancées, en particulier le vieillissement d'un signal. Le vieillissement peut résulter d'une fenêtre glissante sur un signal mono-dimensionnel, classique en filtrage. Il concerne également un signal bi-dimensionnel comme c'est le cas pour la transformée en ondelette (DWT) au fil de l'eau. Le vieillissement est une fonction d'adressage qui intervient entre deux itérations successives d'un algorithme. La Figure IV-8 représente le vieillissement d'un signal mono-dimensionnel (a) et d'un signal bi-dimensionnel (b).

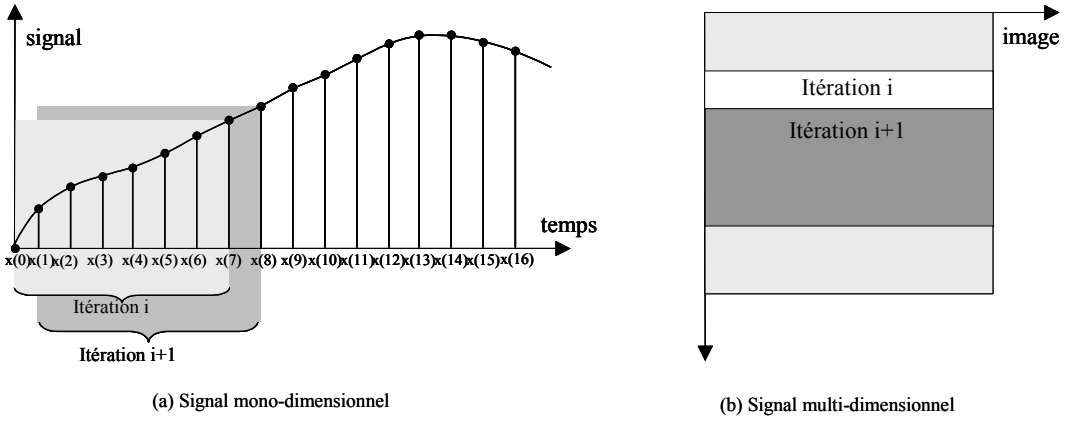


Figure IV-8 : Vieillissement de signaux

Parmi les solutions d'adressage classiquement mises en œuvre sur les processeurs DSP on retrouve soit une recopie des échantillons en mémoire, soit un adressage circulaire. La gestion des échantillons par recopie ou adressage circulaire (pour l'exemple du signal mono-dimensionnel de la Figure IV-8) est illustrée par la Figure IV-9. Une gestion par recopie (a) implique 7 lectures (échantillons *x(1)* à *x(7)*) et 8 ré-écritures (échantillons *x(1)* à *x(8)*) les deux premières itérations. Le parcours des adresses est identique d'une itération à l'autre. Une gestion par adressage circulaire (b) ne nécessite qu'une écriture du nouvel échantillon *x(8)*. Les parcours des adresses évoluent d'une itération à l'autre.

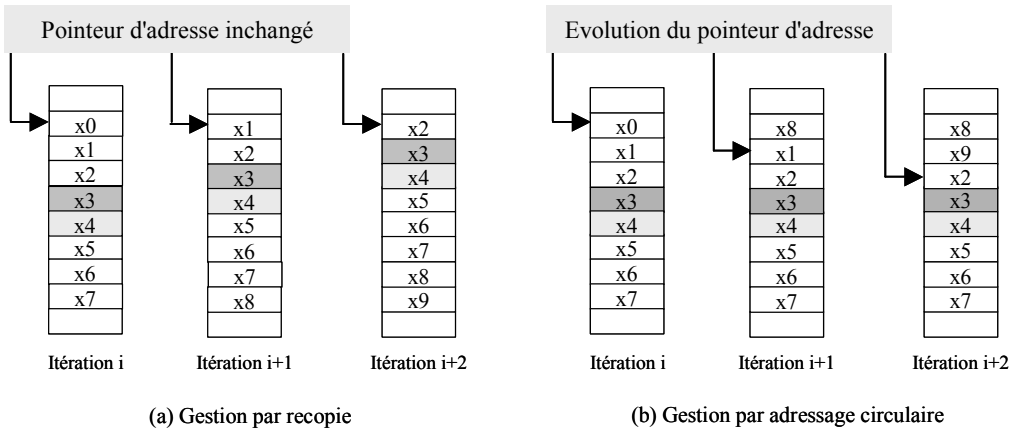


Figure IV-9 : Gestion du vieillissement

- La gestion par recopie utilise la possibilité d'effectuer une lecture et une écriture à la même adresse mémoire durant un cycle d'exécution d'une machine. En disposant d'un registre temporaire dans l'unité de traitement, le cycle d'appel d'un échantillon est complété par un cycle d'écriture à la même adresse que l'échantillon précédemment appelé. Dans ce cas, les appels aux échantillons doivent avoir des indices décroissants. Cette technique offre l'avantage de fournir des adresses fixes pour le vecteur vieillissant. Cependant, elle s'avère coûteuse en consommation et en temps car elle impose une augmentation significative des accès mémoire. Nous ne la retiendrons pas.
- La gestion par adressage circulaire nécessite une évolution des adresses des éléments du vecteur évoluent au cours du temps (entre deux itérations). Cette solution peut facilement être mise en œuvre à l'aide de compteurs modulo L, où L est la taille de la structure adressée. Nous retenons cette solution de gestion d'adressage car elle permet de limiter le nombre d'accès en lecture et en écriture.

Cependant, elle pose une difficulté majeure d'implantation lorsque les accès se font en parallèle (existence de plus d'un banc mémoire) ou lorsque l'architecture est pipeline (recouvrement temporel de plusieurs itérations successives).

Pour répondre à ce problème d'implémentation, nous développons une approche se basant sur un modèle architectural des unités de mémorisation. Les adresses physiques des données d'un vecteur sont invariantes (une donnée est placée en mémoire physique et conserve cet emplacement quelle que soit l'itération de l'algorithme). L'ordre d'accès des échantillons dans le vecteur vieillissant évolue suivant une adresse logique (à chaque itération les adresses logiques sont recalculées). Nous utilisons une représentation sous forme de graphes permettant de modéliser les séquences d'accès aux données en fonction du vieillissement à opérer et de la séquence d'accès au vecteur sur une itération. La gestion des vecteurs vieillissants soulève aussi le problème de la gestion de conflits d'accès aux mémoires sur plusieurs itérations. Lorsqu'il n'y a pas d'évolution des adresses d'une itération à l'autre, le graphe de contraintes mémoire *MCG*

suffit à représenter les conflits d'accès à la mémoire. Ce modèle devra être complété pour permettre de modéliser tous les conflits d'accès aux données d'un vecteur vieillissant quelle que soit l'itération.

Notre approche propose :

-1- un modèle architectural générique de l'unité de mémorisation.

-2- une modélisation du vieillissement

-3- une projection du modèle de vieillissement sur notre architecture

IV.3.2. Modèle architectural générique

Pour pouvoir mettre en œuvre une génération automatique d'une architecture permettant la gestion de la mémorisation des variables de vecteurs vieillissants, nous définissons des adresses logiques dont la séquence évolue et des adresses physiques permettant d'exprimer un invariant.

- Une adresse logique pointe sur une case mémoire contenant l'adresse physique de la variable à accéder (indirection de l'adresse). L'évolution de l'indirection est traitée entre 2 itérations et permet de gérer le vieillissement.
- Une adresse physique pointe sur la case mémoire contenant la valeur de la variable à accéder. L'évolution de l'adresse physique répond à la séquence de transfert.

L'objectif de la définition d'une adresse logique et d'une adresse physique est d'introduire une architecture permettant de gérer une distribution de données de vecteur vieillissant multi-bancs.

L'architecture de mémorisation et d'adressage des vecteurs vieillissants est représentée en Figure IV-10. Les générateurs d'adresses logiques sont des compteurs modulo dont la séquence d'adressage évolue à chaque itération en fonction du glissement de la fenêtre du vecteur vieillissant. Les compteurs génèrent des adresses logiques qui permettent d'extraire à partir d'un espace d'adressage logique commun aux différents compteurs des adresses physiques. Une adresse physique se décompose en un numéro de banc mémoire et une adresse. Elle permet de retrouver les valeurs d'une donnée d'un vecteur vieillissant quelle que soit l'itération.

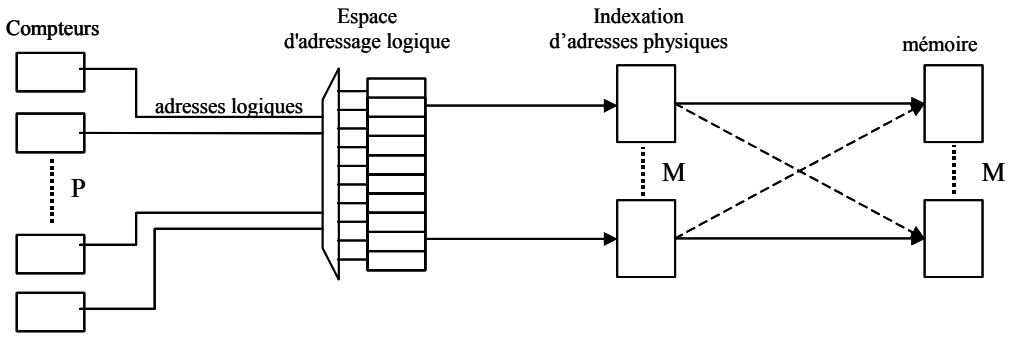


Figure IV-10 : Architecture de mémorisation pour vecteurs vieillissants

IV.3.3. Types de séquences des vecteurs vieillissants

On peut définir deux types de séquences pour un vecteur vieillissant, la séquence de vieillissement des échantillons du vecteur d'une itération à l'autre et la séquence de transfert des échantillons sur une itération.

- Séquence de vieillissement : la séquence de vieillissement est caractérisée par un glissement de la fenêtre sur un certain nombre d'échantillons sur un vecteur de taille infini.

Dans de nombreuses applications de traitement du signal, le vieillissement se résume au décalage d'une valeur sur la suivante entre chaque itération de l'algorithme. On peut exprimer le vieillissement en fonction de l'évolution des variables du vecteur vieillissant ou en fonction de l'évolution des adresses des variables du vecteur vieillissant, les adresses sont dans ce cas les adresses logiques des variables du vecteur vieillissant.

- Séquence d'accès logique : la séquence d'accès logique exprime l'ordre des accès aux variables du vecteur vieillissant sur une itération de l'algorithme. La séquence peut être définie par rapport aux accès spécifiés dans une itération de l'algorithme ou par rapport à des adresses logiques.

IV.3.4. Sémantique

Nous définissons plusieurs graphes permettant de représenter plusieurs types de séquence d'accès. Nous allons définir, dans cette section, la sémantique associée à chaque graphe.

Le graphe de vieillissement exprime le glissement de la fenêtre sur le vecteur vieillissant d'une itération à l'autre. Il représente la séquence de vieillissement.

Le graphe de séquence d'adressage logique représente la séquence d'accès logiques, soit la séquence d'accès aux adresses logiques du vecteur vieillissant sur une itération de la spécification algorithmique. Le graphe de séquence logique est déterminé en aval de l'ordonnancement.

Le graphe de séquences unifiées intègre le vieillissement des adresses logiques et les séquences d'accès aux adresses logiques. Il permet de modéliser toutes les séquences d'accès aux adresses logiques quelle que soit l'itération de la spécification.

Le graphe de conflit d'accès "rotatif" permet de modéliser tous les conflits d'accès aux données du vecteur vieillissant quelle que soit l'itération. Il sert à définir l'architecture mémoire pour une distribution de données multi-bancs d'un vecteur vieillissant.

La séquence de vieillissement des adresses du vecteur, pour une spécification donnée, est déterminée à partir du graphe flot de signaux. Le graphe séquence d'adressage logique est obtenu après ordonnancement des opérations de l'unité de traitement. Le vieillissement d'un

vecteur ainsi que la séquence d'accès aux adresses logiques du vecteur peuvent être modélisés sous la forme d'un seul graphe, le graphe de séquences unifiées. Les conflits d'accès aux données du vecteur vieillissant sont exprimés par un graphe de conflits d'accès "rotatif". A partir de ce graphe une distribution des données du vecteur vieillissant peut être réalisée sur l'architecture mémoire générique spécifiée dans le paragraphe précédent. La génération des adresses est spécifiée par le graphe de séquences unifiées. Le flot de conception des unités mémoires pour les vecteurs vieillissants est résumé par la Figure IV-11.

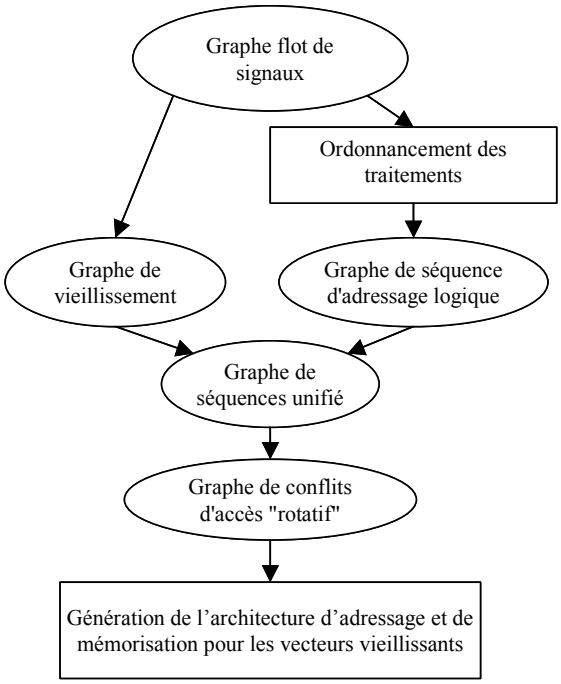


Figure IV-11 : Flot de conception des unités mémoire pour les vecteurs vieillissants.

a. Exemple

Le vieillissement d'un vecteur est exprimé explicitement dans la spécification comportementale de l'application. Le glissement de la fenêtre sur le vecteur exprime le vieillissement du vecteur ; on peut représenter le vieillissement des échantillons du vecteur ou le vieillissement des adresses du vecteur. Nous allons, à travers l'exemple suivant, mettre en évidence toutes les informations nécessaires pour définir une approche formelle permettant d'implanter les vecteurs vieillissants sur une architecture mémoire générique.

```
x[0] : = xnouvel-échantillon
tmp := x[0];
for ( i=1 ; i =3; i++)
        tmp = tmp+ x[i];
End for;
        :
For (i = 3 ; i = 1 ; i--)
        x[i]=x[i-1];
End for;
```

Pour chaque itération de l'algorithme ci-dessus nous représentons l'évolution du vecteur x par rapport au signal d'entrée.

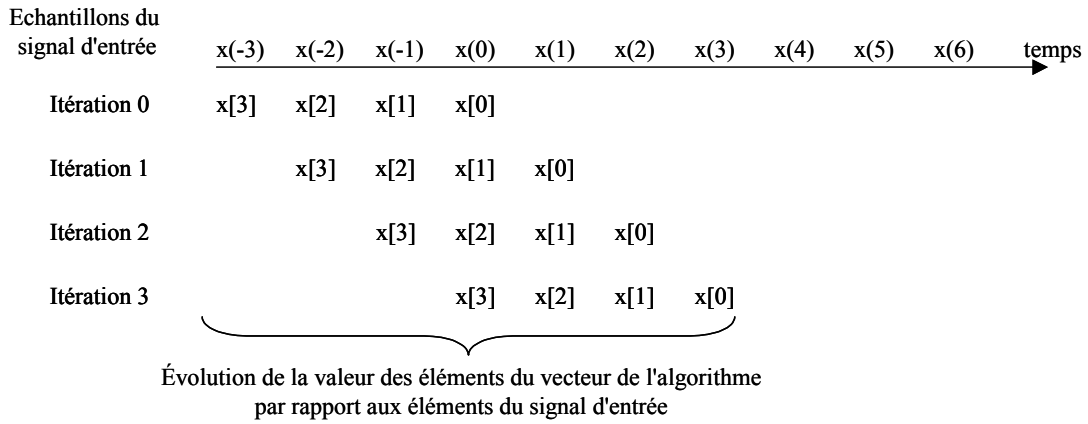


Figure IV-12 : Evolution du vecteur de l'algorithme par rapport au vecteur d'entrée

La séquence d'accès aux éléments du vecteur *x[]* est fixée par l'ordonnancement des opérations et les dépendances de données de l'algorithme. Pour cet exemple, vues les dépendances de données, le vecteur ne peut être accédé que de la manière suivante :

$$x[0] \rightarrow x[1] \rightarrow x[2] \rightarrow x[3]$$

A chaque élément du vecteur de l'algorithme, nous associons une adresse logique qui évoluera entre chaque itération de l'algorithme. Cette évolution permettra de restituer les bonnes valeurs des éléments du signal d'entrée en fonction de l'itération considérée. Nous définissons des adresses logiques *@x[]* pour chaque élément du vecteur vieillissant de l'algorithme. La séquence d'accès aux adresses logiques du vecteur *x[]* est la suivante :

$$@x[0] \rightarrow @x[1] \rightarrow @x[2] \rightarrow @x[3]$$

La séquence d'accès aux données n'est pas forcément régulière, il faut également tenir compte du nombre de cycles entre deux accès aux données pour pouvoir caractériser complètement une séquence d'accès à un vecteur.

Reprenons l'exemple précédent en supposant que le traitement des échantillons s'effectue du plus récent au plus vieux. Pour les itérations successives de l'algorithme, nous définissons la séquence d'accès aux échantillons de l'algorithme, la séquence d'accès aux échantillons du vecteur d'entrée et la séquence d'accès aux adresses associées aux éléments du vecteur d'entrée. A chaque itération l'échantillon le plus vieux est remplacé par le nouvel échantillon. La séquence d'adressage évolue à chaque itération de l'algorithme.

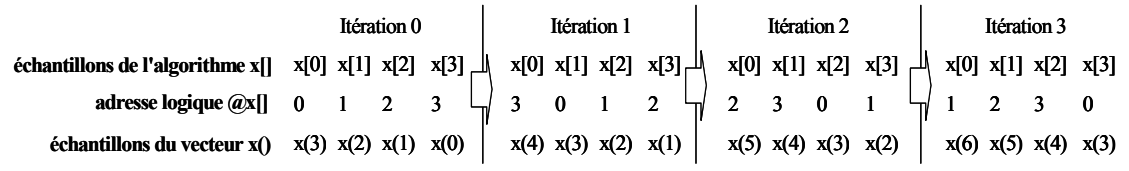


Figure IV-13 : Séquences d'accès

Le vieillissement des échantillons peut être exprimé par rapport aux valeurs des échantillons ou par rapport à l'évolution des adresses logiques du vecteur.

$$x[i] = x[i-1]$$

$$\text{ou } @x[i] = @x[i-1] \% L$$

avec *L* est la longueur de la fenêtre sur les vecteur vieillissant.

La séquence d'accès sur une itération et le vieillissement entre deux itérations sont facilement modélisables à l'aide de graphes. Pour pouvoir implanter les vecteurs vieillissants sur une architecture mémoire générique il faut trouver une représentation unique pour le vieillissement et les séquences d'accès aux éléments du vecteur, il faut également modéliser les conflits d'accès aux données quelle que soit l'itération de l'algorithme. Les graphes permettant de modéliser les séquences et les conflits d'accès à un vecteur vieillissant sont présentés ci-dessous.

b. Graphe de séquence d'adressage logique

Lorsque l'ordonnancement des traitements est effectué, nous pouvons extraire la séquence d'accès en lecture et en écriture de toutes les variables appartenant à un vecteur vieillissant et cela pour une itération de l'algorithme. Nous pouvons modéliser la séquence d'accès aux variables d'un vecteur vieillissant à l'aide d'un graphe (Figure IV-14).

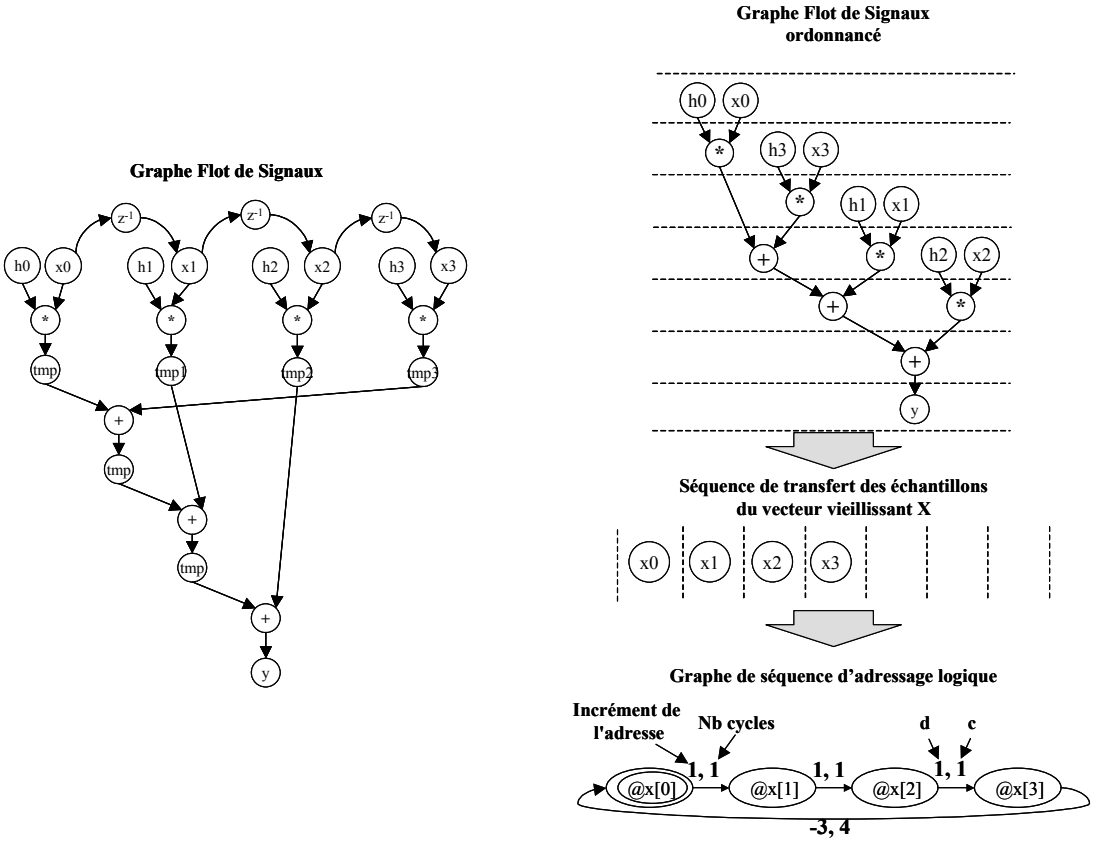


Figure IV-14 : Expression de la séquence d'accès aux adresses logiques sur une itération

Définition : soit un graphe de Séquence d'Adressage Logique *GSAL(V, E, w)*, un graphe cyclique, orienté, et pondéré qui représente la séquence d'accès aux adresses logiques du vecteur vieillissant. Soit *V* l'ensemble des adresses logiques d'un vecteur vieillissant, la cardinalité de l'ensemble |*V*| est définie par la taille de la fenêtre du vecteur vieillissant. Les arcs *E* représentent la succession de transferts des différents échantillons du vecteur vieillissant. Un arc *Ei,j* existe si l'accès au nœud *Vj* est consécutif à l'accès au nœud *Vi*. Les arcs sont pondérés par un couple *w(d,c)* où *d* représente l'évolution de l'adresse entre deux accès consécutifs et *c* le nombre de cycles entre deux accès consécutifs.

c. Graphe de vieillissement

Le vieillissement d'une adresse logique s'exprime facilement d'une itération à l'autre. Soit $@x[j]^i$ une adresse logique quelconque, *i* une itération de l'algorithme, l'adresse logique de l'itération suivante $@x[j]^{i+1}$ s'exprime en fonction de l'adresse de l'itération courante $@x[j]^i$, de la valeur du glissement de la fenêtre *k* et de la longueur de la fenêtre du vecteur vieillissant *L*.

$$@\mathrm{x[j]}^{i+1} = (@\mathrm{x[j]}^{i} - k)\%\, L$$

Le vieillissement peut se représenter sous la forme d'un graphe, le Graphe de Vieillissement des adresses logiques, *GVA*.

Définition : soit le Graphe de Vieillissement des Adresses Logiques *GVA (V, E, k)* un graphe orienté et pondéré où *V* est l'ensemble des nœuds du graphe tel qu'un nœud *j* est une adresse logique d'un élément du vecteur vieillissant à l'itération *i*. *E* est l'ensemble des arcs représentant une relation de vieillissement entre deux itérations de l'algorithme. Les arcs sont pondérés par un poids *-k* représentant le glissement de la fenêtre de l'itération courante à l'itération suivante. Les adresses logiques sont exprimées modulo la longueur de la fenêtre de vieillissement (en nombre d'échantillons).

Simplification : le graphe peut se simplifier en limitant le nombre d'éléments d'un graphe à deux nœuds et un arc pondéré par le glissement. Les nœuds représentent les adresses logiques d'un même accès sur deux itérations successives.

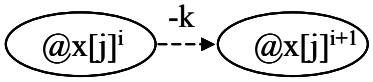


Figure IV-15 : Graphe de vieillissement

Nous utiliserons le modèle de graphe de séquence simplifié pour alléger les représentations.

d. Graphe de séquences unifiées

Pour simplifier la manipulation de graphes, il faut pouvoir exprimer le vieillissement du vecteur et la séquence d'adressage du vecteur sur une itération. Pour cela, nous développons, à

partir du graphe de vieillissement des adresses logiques et du graphe de parcours algorithmique des adresses logiques du vecteur vieillissant, un graphe de séquences d'adressage unifiées. Ce graphe peut se construire également à partir de l'expression explicite du vieillissement de la spécification comportementale et du rapport sur les accès mémoire fourni après ordonnancement des opérations.

Définition : un graphe de séquences unifiées, noté *GSU(V, E, w')*, est un graphe cyclique, orienté et pondéré.

Les nœuds *V* représentent les adresses logiques du vecteur vieillissant. Les arcs *Eij* représentent la séquentialité entre *Vi* et *Vj*. Les arcs sont pondérés par *w'*, un triplet (*d,c,e*) où *d* représente l'évolution de l'adresse entre deux accès consécutifs et *c* le nombre de cycles entre deux accès consécutifs et e représente l'offset d'adresse logique entre deux itérations de l'algorithme. Le graphe contient alors les informations temporelles indiquant à quel cycle mémoire les données sont accédées.

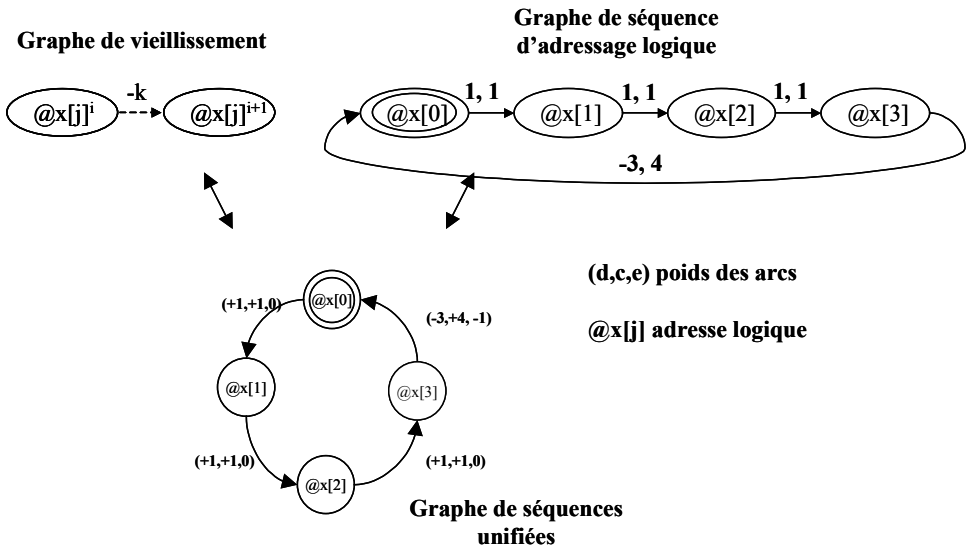


Figure IV-16 : Graphe de séquences unifiées

La Figure IV-16 représente la construction du graphe de séquences unifié à partir des graphes de vieillissement et de séquence d'adressage logique.

Nous pouvons extraire des règles de construction pour les graphes de séquence unifiées.

Règle 1 : il existe un arc *Eij* entre les nœuds *Vi* et *Vj* si et seulement si le nœud *Vj* est le prochain nœud accédé après l'accès à *Vi* dans la séquence d'accès aux adresses logiques.

Règle 2 : la somme des poids d est nulle pour tout cycle du graphe *GSU(V, E;w')*.

Règle 3: le poids *e* de l'ensemble des arcs E est nul sauf celui de l'arc reliant un nœud au nœud origine de la séquence. Le poids *e* de cet arc est égal à la valeur du glissement *k* sur le vecteur vieillissant multipliée par le nombre de tranche de pipeline de l'architecture.

Règle 4: l'introduction de jetons est nécessaire pour modéliser les séquences d'accès pour les architectures pipeline. Il y a autant de jetons que de tranches de pipeline.

Règle 5: le nombre d'espaces mémoire nécessaires à garantir la cohérence des données définit le modulo à appliquer pour les calculs des adresses.

Pour illustrer l'influence des architectures pipelines sur la construction du GSU, revenons sur l'exemple précédent. Lorsque la séquence d'accès n'est pas pipeline, le chronogramme des accès reprend la séquence d'accès de la Figure IV-17.

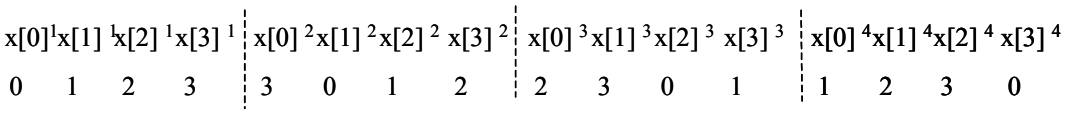


Figure IV-17 : séquence d'accès non pipeline

Pour obtenir la suite des adresses logiques auxquelles aller chercher les données nécessaires au traitement, il suffit de faire circuler un jeton à l'intérieur du *GSU* (cf. Figure IV-18). La valeur initiale de l'adresse pour ce jeton dépend de l'itération à laquelle on se trouve. Elle vaut 0 pour l'itération 1 par exemple et est repérée 0 sur la Figure IV-18. On vérifie en faisant circuler ce jeton suivant le sens des arcs, que la suite des adresses calculées est 0, 1, 2, 3 puis 3, 0, 1, 2 puis 2, 3, 0, 1 etc…).

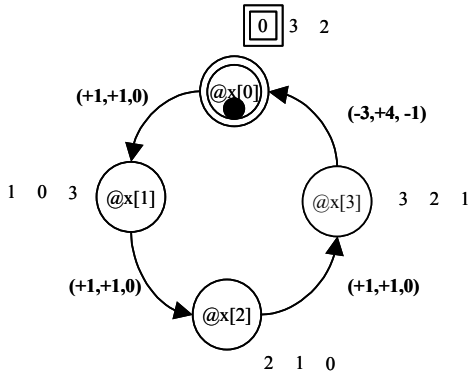


Figure IV-18 : GSU, accès non pipelines

Lorsque l'architecture est constituée de tranches de pipeline, deux tranches pour notre exemple, le chronogramme des accès se replie comme indiqué sur la Figure IV-19.

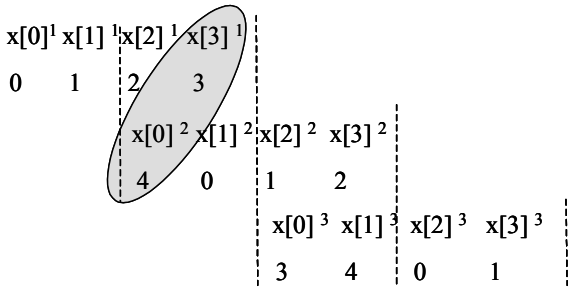


Figure IV-19 : séquence d'accès avec 2 tranches de pipeline

Pour cet exemple, il faut ajouter un espace mémoire pour garantir la cohérence des données. En effet, si on conserve la séquence d'accès de l'architecture non pipeline, la donnée $x[3]^1$ sera écrasée par la donnée $x[0]^2$. le modulo servant à déterminer les adresses logiques est maintenant de 5. Le GSU doit également porter 2 jetons puisque sont réalisés en même temps 2 accès aux données pour deux itérations successives (voir Figure IV-20).

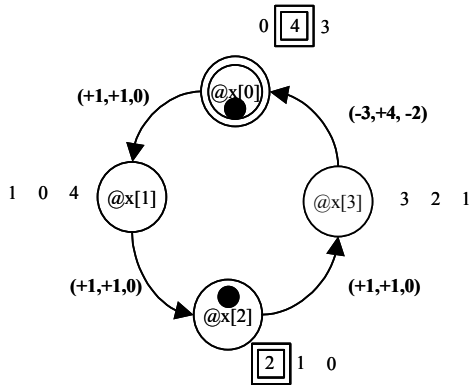


Figure IV-20 : GSU, accès pipelines

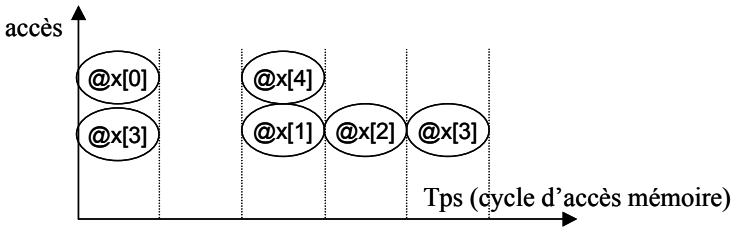
Par exemple, on doit accéder simultanément à la donnée contenue à l'adresse de x[2] de l'itération 1 ($@x[2]^1$) et à la donnée contenue à l'adresse de x[0] de l'itération 2 ($@x[0]^2$). La valeur initiale pour chacun de ces deux jetons dépend des itérations où l'on se trouve. Pour notre exemple, si l'on se trouve aux itérations 1 et 2, elle vaudra 2 pour l'itération *i* et 3 pour l'itération *i+1*, avec i = 1.

Lorsque le jeton pour l'itération i passe du sommet @x[3] au sommet @x[0] (arc entrant dans le nœud origine), l'adresse calculée n'est pas celle pour l'itération *i+1*, puisque celle –ci s'obtient avec l'autre jeton. Cette adresse est en fait l'adresse de la première donnée du vecteur vieillissant accédée à l'itération *i+2*. le poids c de l'arc n'est pas simplement le glissement de la fenêtre (*–k*) mais le produit du glissement par le nombre de tranches de pipeline intégrées à l'architecture (*-k * NbT*). En faisant circuler les deux jetons, on vérifie bien que les séquences d'accès obtenues correspondent au chronogramme des accès.

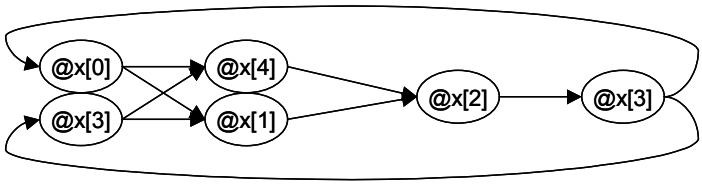
Le graphe de séquences unifiées permet de définir les adresses logiques des vecteurs vieillissants quelle que soit l'itération, le nombre de tranches de pipeline et le nombre d'accès parallèles aux données.

e. Notion de cycles, choix du parcours des adresses logiques.

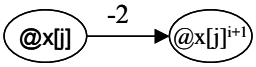
Lorsque les accès ne sont pas exclusivement séquentiels et que la distribution des données n'est pas connue, le graphe de séquences unifiées fait apparaître des cycles. Dans l'exemple suivant nous étudierons le parcours des différents cliques permettant de recouvrir tous les accès aux données d'un vecteur vieillissant tout en réduisant la complexité de l'adressage.



Chronogramme des accès



Graphe 1 : séquences d'accès possibles



Graphe de vieillissement

Figure IV-21 : Séquence accès et vieillissement

Le chronogramme des accès représente une séquence donnée aux adresses logiques. Le graphe 1 illustre les possibilités de séquences d'accès pouvant être réalisées. Sur ce graphe on distingue 4 séquences d'accès possibles, soient quatre cycles d'accès distincts.

- @x[0], @x[4], @x[2], @x[3] (1)
- @x[0], @x[1], @x[2], @x[3] (2)
- @x[3], @x[4], @x[2], @x[3] (3)
- @x[3], @x[1], @x[2], @x[3] (4)

Le glissement introduit par le vieillissement est de 2, il est modélisé par le graphe de vieillissement.

Les quatre séquences se retrouvent sur le graphe de séquences unifiées, et forment quatre cycles. Le parcours des différents cycles permet de déterminer les cycles recouvrant toutes les adresses logiques. Dans l'exemple de la Figure IV-22, le nombre minimum de cycles recouvrant tous les nœuds du graphe est de 2. Nous pouvons dans ce cas déterminer qu'il faut deux générateurs d'adresses logiques et deux bancs pour pouvoir accéder à toutes les données du vecteur vieillissant. Sur la Figure IV-22, nous faisons apparaître deux séquences distinctes, Ces séquences vont permettre la définition du graphe de séquences unifiées. Le choix des cycles est guidé par la limitation de la complexité de la séquence d'adresse logique. Les incréments de l'adresse logique sont +4, -2, +1 pour le cycle (1), +1, +1, +1 pour le cycle (2), +1, -2, +1 pour le cycle (3), -2, +1, +1 pour le cycle (4). Le cycle (2) permet de réaliser un incrément de un entre

chaque accès, on conserve cette séquence d'accès. Il reste à accéder aux adresses logiques @x[3] et @x[4], le cycle (3) permet d'effectuer la séquence d'accès @x[3], @x[4]. Les deux cycles retenus sont le cycle (2) et le cycle (3) où seuls les accès aux adresses logiques @x[3] et @x[4] seront effectués.

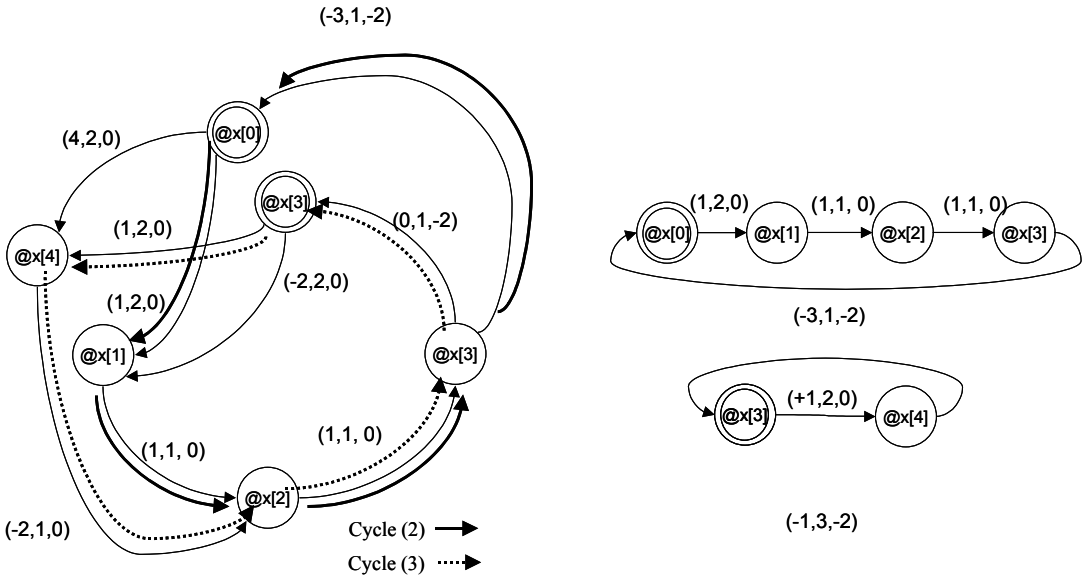


Figure IV-22 : Graphes de séquences unifiées, choix des cycles

f. Graphes de conflit d'accès "rotatif"

Pour éviter les conflits générés par le pipeline il faut trouver une représentation mettant en évidence tous les conflits d'accès mémoire. Pour cela nous définissons un graphe de conflit d'accès "rotatif". Les conflits d'accès peuvent être exprimés d'une part avec le vieillissement des variables du vecteur, et d'autre part avec la séquence d'accès aux adresses logiques. On construit d'abord un graphe de conflit pour une itération de l'algorithme puis on ajoute les arcs de conflit qui dépendent du vieillissement et des tranches de pipeline en faisant "tourner" les arcs de conflit du graphe en fonction du glissement de la fenêtre vieillissante. Le terme "tourner" signifie que s'il existe un arc *E* entre *Vi* et *Vj*, alors il a un arc entre *Vi+k* et *Vj+k*, *k* représentant le glissement de la fenêtre temporelle entre deux itérations.

Définition : un graphe de conflit d'accès "rotatif" note *GCA(V, E)* permet d'exprimer tous les conflits d'accès à un vecteur vieillissant. Un nœud *Vi* représente une adresse (logique ou physique) nécessaire à la mémorisation du vecteur sans perte d'information. Un arc *Eij* représente un conflit d'accès entre les nœuds *Vi* et *Vj*.

Règle 1 : le nombre de nœuds du graphe correspond au nombre d'adresses à accéder quelles que soient les itérations.

Règle 2 : la rotation des arcs s'effectue dans le sens du vieillissement du vecteur et en fonction du glissement de la fenêtre.

Revenons sur l'exemple précédent. Nous avons constaté que le fait de pipeliner l'architecture impliquait des accès simultanés à la mémoire et, dans certain cas, l'ajout d'espace mémoire. Etendons le chronogramme des accès à 6 itérations de notre algorithme (Figure IV-23). Les accès concurrents apparaissent sur ce chronogramme. Observons les accès sur différentes plages de chevauchement des tranches de pipeline.

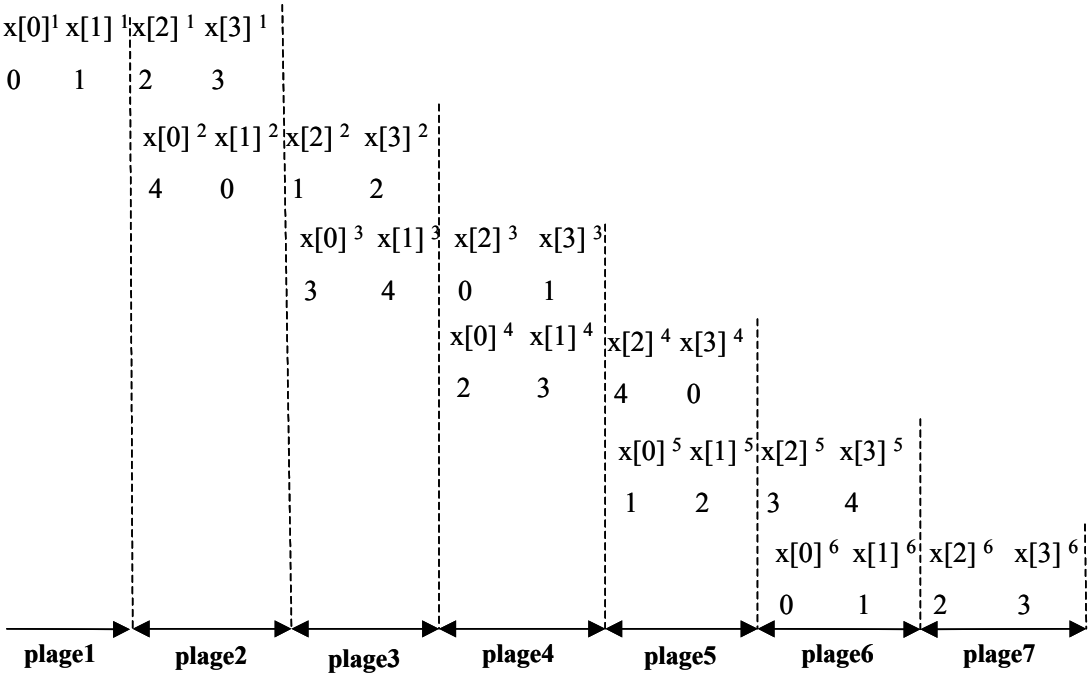


Figure IV-23 : chronogramme des accès pipeline sur 6 itérations

Dans la plage 2, les accès concurrents se produisent pour les adresses logiques 2 et 4, puis 3 et 0. Sur la plage 3, ce sont successivement les adresses logiques 1,3 et 2,4 qui sont concurrentes. Les adresses logiques pour lesquelles des accès concurrents existent sont dîtes en conflits. On construit facilement un graphe de conflit à partir des observations précédentes. Le graphe de conflits possède autant de nœuds qu'il y a d'adresses logiques accédées, dans notre exemple 5 nœuds. Un arc marque un conflit d'accès à deux adresses logiques pour chaque plage. Pour la plage 2, nous pouvons modéliser les conflits par le graphe de la Figure IV-24.

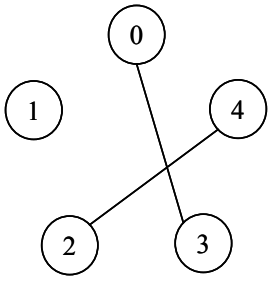


Figure IV-24 : graphe de conflit pour la plage 2

Pour chaque plage, nous pouvons construire un graphe de conflit. Sur la Figure IV-25, les graphes de conflits des plages 3, 4, 5, 6 sont successivement représentés.

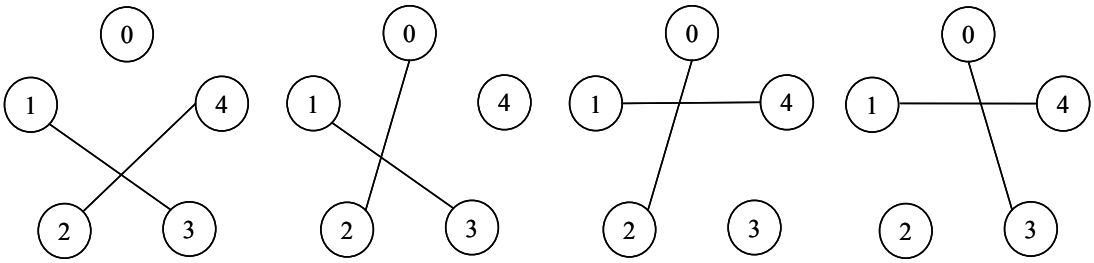


Figure IV-25 : graphe de conflits des plages 3, 4, 5, 6

En superposant les différents graphes de conflits, nous obtenons le graphe de conflit d'accès "rotatif" qui modélise tous les conflits d'accès aux données d'un vecteur vieillissant quelle que soit l'itération de l'algorithme comme le montre la Figure IV-26.

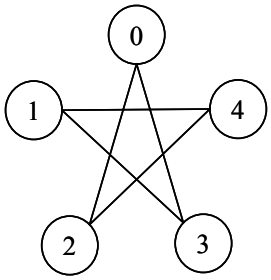


Figure IV-26 : graphe de conflit d'accès rotatif

Ce graphe de conflit d'accès "rotatif" peut être construit à partir du graphe de séquences unifiées, *GSU*. Les nœuds peuvent déterminer à partir de toutes les adresses logiques définit sur le *GSU*. Les arcs sont construits en observant la position des jetons sur le graphe à un instant donné, puis en décalant les arcs suivant le vieillissement.

Nous pouvons également exprimer par un graphe de compatibilité d'accès rotatif où les arcs représentent la possibilité de positionner deux éléments d'un vecteur vieillissant dans un même banc mémoire. Le graphe de compatibilité d'accès "rotatif" permet, en dénombrant le nombre de cliques, de déterminer le nombre minimum de bancs mémoire à implémenter pour garantir une gestion correcte des vecteurs vieillissants. La Figure IV-27 représente le graphe de compatibilité de notre exemple.

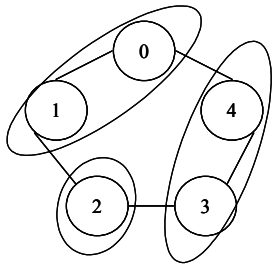


Figure IV-27 : Graphe de compatibilité d'accès rotatif

Pour cet exemple, il faudra au minimum trois bancs mémoires pour garantir la cohérence des données du vecteur vieillissant. L'architecture mémoire permettant de gérer les accès aux vecteurs vieillissants dans le cas d'une architecture de traitement pipeline est décrite dans la Figure IV-28.

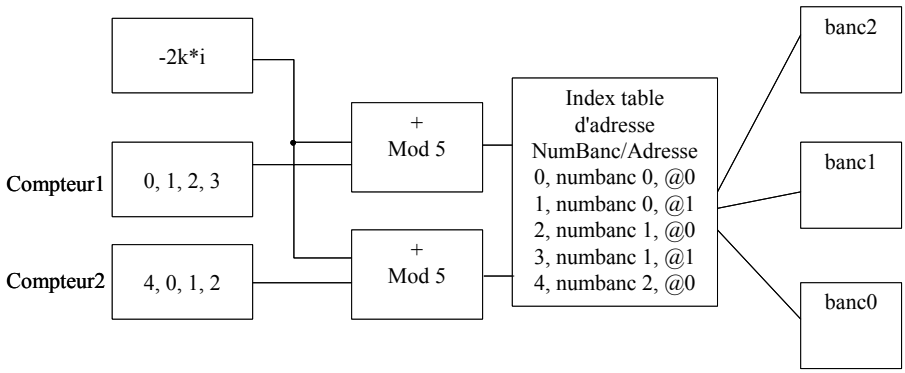


Figure IV-28 : Architecture mémoire pour le vieillissement

Les compteurs 1 et 2 permettent de générer les séquences d'adresses logiques sur deux itérations. La somme modulo 5 permet de définir la valeur de l'adresse logique accédée suivant l'itération en fonction du vieillissement et du nombre de tranches pipeline. Les données sont accédées suivants la table d'indexation dans les différents bancs mémoire.

IV.3.5. Exemples

Dans ce paragraphe nous allons illustrer la validité et l'intérêt du formalisme et de l'approche automatique de gestion de vecteurs vieillissants. Un premier exemple introduit le parallélisme d'accès, le second proposera une modélisation d'une séquence d'accès parallèle en mode pipeline à un vecteur vieillissant.

a. Gestion des accès parallèles à un vecteur vieillissant

Les différents modèles de graphes sont représentés dans les Figure IV-29 et Figure IV-30 et illustrent la modélisation d'accès réalisés en parallèle. Supposons que l'étape d'ordonnancement amène à la séquence d'accès aux éléments du vecteur vieillissant suivante.

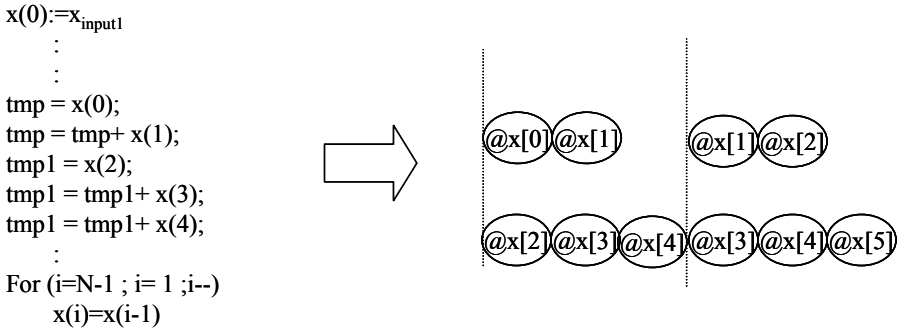


Figure IV-29 : Séquence d'accès parallèles

A partir de cet ordonnancement, nous pouvons déterminer les graphes de vieillissement, d'adressage logique et de séquences unifiées. Le parallélisme d'accès est représenté par deux graphes.

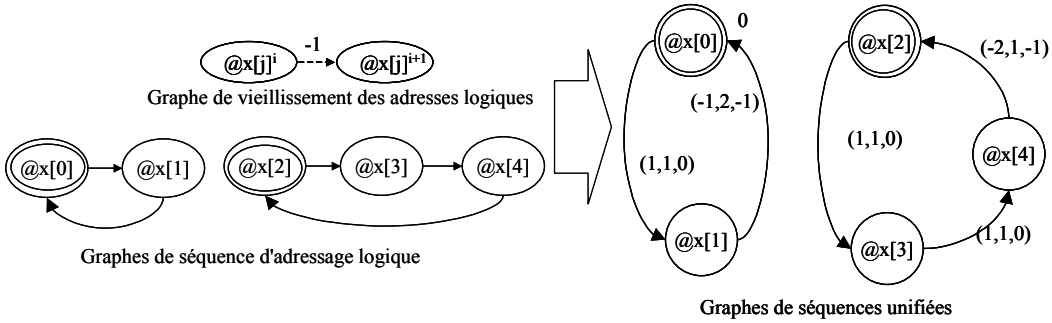


Figure IV-30 : Graphes de séquences pour accès parallèles

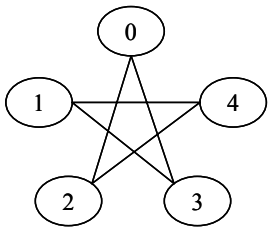


Figure IV-31 : Graphe de conflits rotatif

Pour garantir la cohérence des données lors des accès au vecteur vieillissant, les données devront être distribuées sur trois bancs mémoire. Le graphe de séquences unifiées permet de définir les séquences d'adressages aux différents bancs mémoire.

b. Gestion des accès parallèles à un vecteur vieillissant pour une architecture pipeline

Les différents modèles de graphes sont représentés dans la Figure IV-32, pour illustrer la modélisation d'accès réalisés en parallèle sur une architecture pipeline. Supposons que l'étape d'ordonnancement amène à la séquence d'accès aux éléments vecteur vieillissant suivantes.

Les accès mémoire pipeline sont ici représentés par la présence de jetons dans les graphes de séquence qui indiquent la valeur des adresses en fonction de l'étage de pipeline de l'architecture

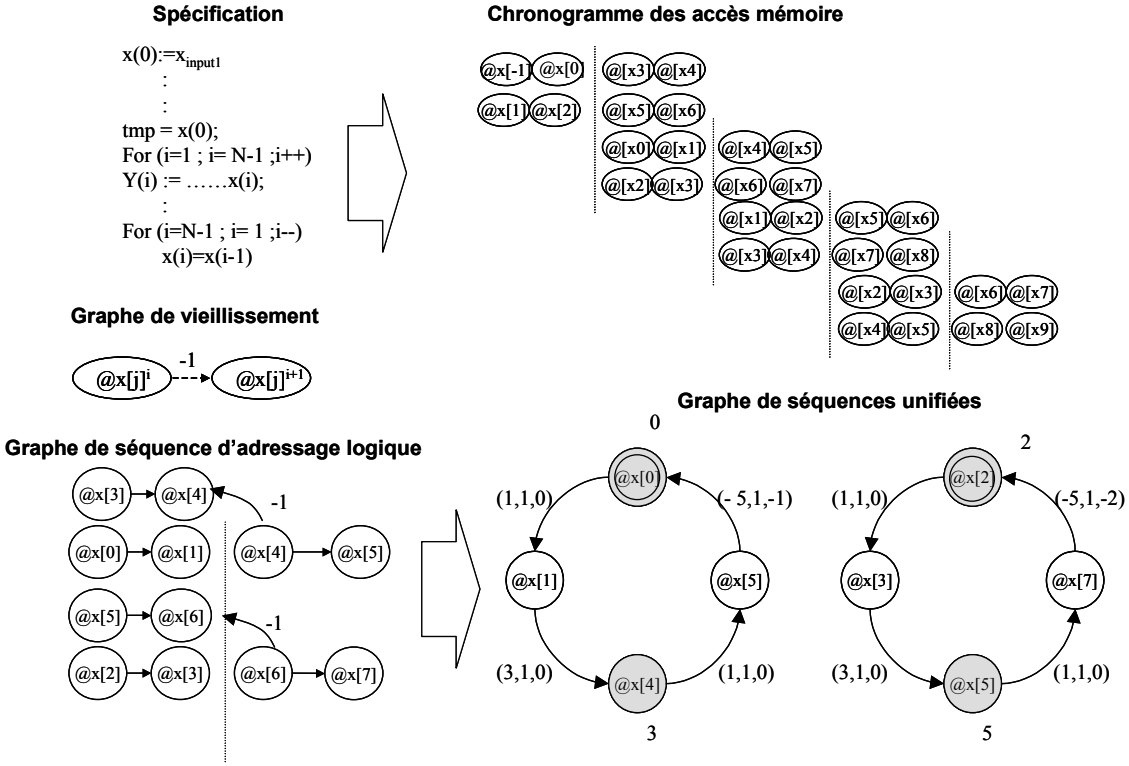


Figure IV-32 : Accès parallèle en pipelines

IV.4. Conclusion

En introduisant des modèles à base de graphes, nous avons dans un premier temps, réaliser un ordonnancement sous contraintes de distribution et de placement des données en mémoire. Cet ordonnancement peut être amélioré en anticipant les accès aux données, le chapitre suivant développe une approche améliorant ses performances. Dans un second temps, les modèles formels nous ont permis de répondre aux problèmes de gestion des accès aux données de vecteurs vieillissants utilisés dans les applications TDSI. Nous nous sommes intéressés à la gestion de la mémorisation des vecteurs vieillissants car ils apparaissent dans de nombreuses applications. Les graphes et les modèles architecturaux présentés permettent de gérer des accès **parallèles** et **pipelines** à ce type de donnée.

Chapitre V
Gestion des accès mémoire par flux tiré

La mémoire forme un goulet d'étranglement pour les accès aux données ; l'optimisation de l'ordonnancement durant la synthèse nécessite d'anticiper les accès aux données à traiter. En nous basant sur les propriétés de flux des données des applications de traitement du signal, nous avons développé une méthode innovante pour la gestion des accès mémoire. Cette méthode s'inspire de la méthode des flux tirés très utilisée dans le domaine de l'organisation et de la gestion de production. Une fois adaptée, elle s'utilise dans la phase d'ordonnancement de la synthèse de haut niveau et permet de gérer les accès à la mémoire avec efficacité.

V.1. Introduction

L'ordonnancement des opérations, lors de la synthèse d'architecture, répond à deux contraintes majeures :

- la contrainte de précédence où l'ordonnancement d'une opération suppose la fin de l'exécution de celles dont elle dépend.

- la contrainte de disponibilité des données à traiter. Cette seconde contrainte peut être difficile à lever lorsqu'un ensemble d'opérations ordonnancées à une même date provoque une congestion d'accès à la mémoire.

La mise en œuvre d'un ordonnancement contraint uniquement par la précédence peut amener à une congestion d'accès à la mémoire. La prise en compte de la disponibilité des données permet de réguler les accès à la mémoire. La vérification de la disponibilité peut conduire à une augmentation de la latence de l'architecture puisque que les opérations dont les données ne sont pas disponibles devront être retardées (Figure V-1). Dans ce cas, le parallélisme potentiel de calcul n'est pas exploité et est limité par le parallélisme d'accès aux données en mémoire.

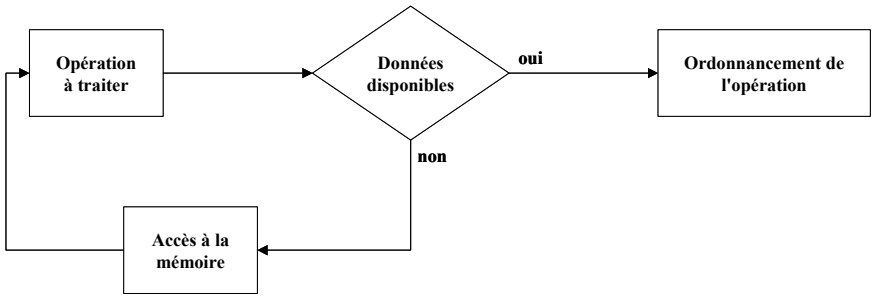


Figure V-1 : ordonnancement d'une opération sous contrainte de disponibilité

La régulation du flot d'accès mémoire nécessite d'anticiper au mieux les besoins avant même que l'ordonnancement des opérations ne l'impose. La Figure V-2 illustre l'introduction de l'anticipation des accès aux données permettant de limiter le nombre d'opérations qui seront retardées. Les accès aux données vont être anticipés et les données utiles aux opérations à ordonnancer seront placées en registres internes de l'unité de traitement.

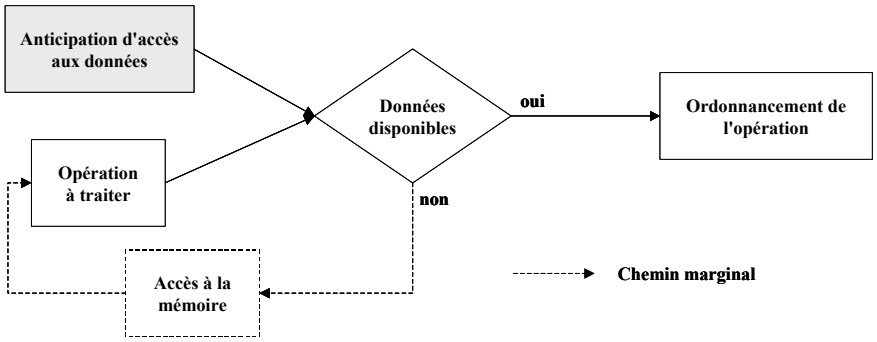


Figure V-2 : ordonnancement d'une opération avec anticipation des accès

Le corollaire de la régulation des accès aux données (lectures) par anticipation est qu'elle ne doit pas conduire à une augmentation prohibitive des registres internes. Nous avons développé une méthode de régulation des flux mémoire inspirée du domaine de la gestion de production par flux tiré. Nous adaptons la méthode kanban à la gestion du flux d'accès entre unité de mémorisation et unité de traitement.

L'objectif de cette méthode est d'anticiper les accès aux données placées en mémoire au plus juste. Si on anticipe trop d'accès aux données, alors le nombre de registres internes à l'unité de traitement va être important et entraîner un surcoût matériel ; si on anticipe trop peu d'accès aux données, le faible nombre de données disponibles peut entraîner un retard dans l'ordonnancement de certaines opérations.

Dans un premier temps nous présentons la gestion de production par kanban puis nous adapterons cette technique à la gestion des lectures et des écritures mémoire pour l'ordonnancement des opérations de l'unité de traitement.

V.2. Gestion par flux tiré

V.2.1. Introduction

L'un des thèmes centraux de la gestion de la production est la maîtrise des flux, flux de matière ou flux d'information. On conçoit aisément qu'un poste opérateur (ou une machine) est correctement géré s'il n'y a pas de rupture d'approvisionnement. Cependant les économies de coût, liées en particulier au coût de stockage, nécessitent une parfaite gestion des stocks d'en cours autour de chaque opérateur ou machine. Nous assimilerons par la suite les machines aux opérations de l'algorithme et les matières aux données à traiter.

La gestion par kanban est une gestion de production à stock zéro, mise au point au Japon par les usines Toyota. Le mot kanban serait apparu dans les chantiers navals japonais qui, dans les années 60, demandaient aux aciéries des livraisons tous les 3 jours au lieu d’une par mois. La méthode industrielle a été mise au point plus récemment chez Toyota avec le concours de Taiichi Ohno [Ohno78], [Ohno90], qui avait observé dans les supermarchés les employés renouveler sur les présentoirs les denrées périssables au fur et à mesure de la demande. En japonais Kanban signifie étiquette, fiche, carte [Sout91].

Le modèle de gestion par kanban est fondé sur :

- l’emploi optimal du personnel,
- la réduction des stocks (par commande journalière),
- la concertation entre la production et la vente pour assurer une charge constante,
- la conception des produits en vue de faciliter leur fabrication,
- la gestion centralisée,

Cette méthode est actuellement l'une des plus utilisées dans le domaine de la gestion de la production industrielle [Jave03].

V.2.2. La méthode kanban

a. Principe

Chaque poste opérateur contient un ensemble de pièces d'en-cours (ou de matière première). La consommation de ces pièces, pour la fabrication réalisée sur ce poste, provoque une demande de matières premières à destination des postes ou stocks en amont. Si le stock de produits finis est plein, la fabrication cesse sur ce poste jusqu'à ce qu'un poste opérateur ou un stock aval classe la demande de ces produits finis.

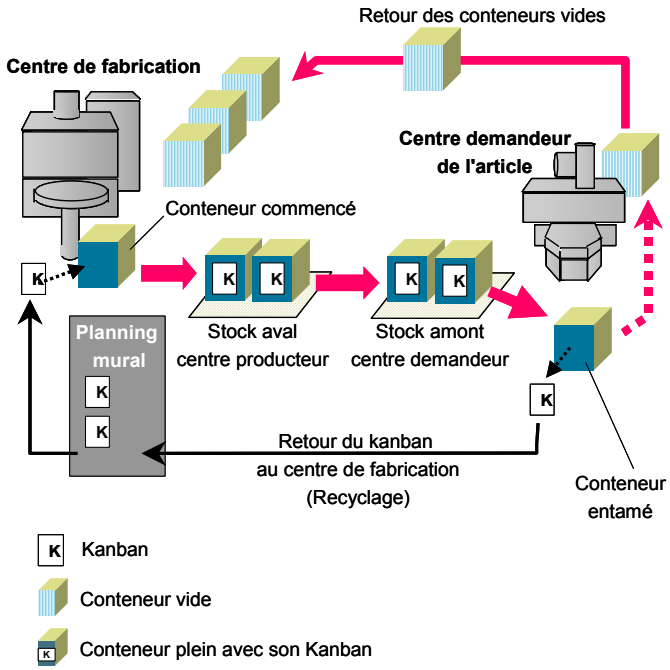


Figure V-3 : Principe de la gestion kanban en production

La méthode des flux kanban est une méthode de gestion à flux tiré (c'est "l'aval" qui sollicite "l'amont") et à stock nul ou minimal.

b. Méthodologie

La méthodologie d'enchaînement de production par Kanban s'établit de la manière suivante ; pour chaque référence travaillée et pour chaque poste opérateur du processus de fabrication, il faut :

- Collecter les données relatives au flux à organiser en caractérisant le flux, le poste amont, le poste aval et la liaison entre les deux postes.

- Définir les paramètres de fonctionnement suivants, la capacité et le nombre de machines par poste, la capacité des conteneurs (lot mini de transfert), la taille du lot mini de fabrication autorisant un lancement (position de l'index gris, cf. Figure V-4), la taille de l'en-cours mini (position de l'index noir, cf. Figure V-4) et la taille du tampon de régulation.
- Mettre en œuvre en confectionnant le planning d'ordonnancement, en définissant le contenu des kanbans, les règles de circulation des kanbans et de fonctionnement du planning
- Affiner le planning en réglant les index en fonction de l'évolution du système et en améliorant l'écoulement du flux.

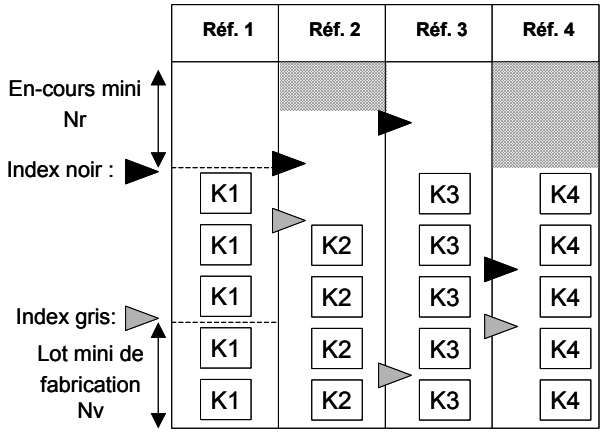


Figure V-4 : Planification kanban

V.3. Gestion kanban des unités de mémorisation

La méthode Kanban est applicable à des productions de type " masse " pour lesquelles le nombre de références n'est pas trop élevé et la demande régulière ou à faibles variations. Nous retrouvons ces propriétés dans les applications de traitement du signal et de l'image et dans les unités de mémorisation qui vont être spécifiées en transposant le modèle de gestion kanban à la gestion des accès mémoire dans la synthèse nous pouvons garantir que:

- le nombre de références (le nombre de registres) allouées ne sera pas trop élevé pour garantir une unité de mémorisation relativement peu coûteuse.
- la demande (le nombre d'accès par cycle) sera régulière ou à faible variation puisque l'anticipation des accès à la mémoire permet de réguler le flot de données.

L'introduction de kanban doit, à partir du modèle matériel de l'unité de traitement, proposer une gestion de l'ordonnanceur avec anticipation des accès aux données.

Le modèle matériel doit répondre à deux problèmes. D'un coté, il doit permettre de stocker toutes les données dont les accès ont été anticipés. Il faut augmenter le nombre de registres

internes de l'unité de traitement. De l'autre, le coût en surface et en consommation de l'unité de traitement doit être maîtrisé. Il faut limiter le nombre de registres de l'unité de traitement.

La gestion de l'ordonnanceur doit permettre de réguler le nombre de données présentes dans l'unité de traitement dont les accès à la mémoire ont été anticipés. Pour ce faire nous introduisons une gestion des lectures et des écritures en mémoire. Les files et listes qui seront définies dans la section V.3.3. servent uniquement à la gestion de l'ordonnanceur et ne sont pas implantées matériellement dans l'unité de traitement.

Le modèle matériel de l'unité de traitement permettant de stocker les données anticipées contient exclusivement des registres. La taille définie pour les différentes files et listes de gestion de l'ordonnanceur permettra de déterminer le nombre maximum de registres alloués pour stocker les données dont les accès ont été anticipés.

Nous adaptons les paramètres définissant la gestion par kanban en production pour la gestion des lectures et des écritures dans le cœur de synthèse de l'outil *GAUT*. Nous allons étudier quels sont les paramètres à mettre en œuvre pour gérer le transfert des données de la mémoire vers le traitement (lecture) et le transfert des données du traitement vers la mémoire (écriture).

Soit M, le nombre de mémoires allouées, pour chaque mémoire i, nous définissons les paramètres suivants.

- Capacité d'un conteneur : un conteneur contient le nombre de données qu'une mémoire peut fournir en un cycle d'accès mémoire. La taille dépend des paramètres de la mémoire allouée. Par exemple, si les mémoires allouées sont des mémoires simples ports, alors la taille $C(i)$ d'un conteneur sera de une donnée. On peut par ailleurs spécifier une taille de conteneur pour les lectures $C_L(i)$ et une taille de conteneur pour les écritures $C_E(i)$.
- Taille du lot minimal de fabrication : $Nv_L(i)$, $Nv_E(i)$, le nombre minimal de kanbans, déclenche respectivement, l'activation de l'ordonnancement des opérations de l'unité de traitement et l'écriture des données produites par l'unité de traitement.
- Temps de rotation minimal d'un kanban : le temps de rotation se limite au temps d'accès à la mémoire en lecture ou en écriture.

  $Tr_L(i)$ = temps d'accès à la mémoire(i)
- Taille de l'en-cours minimal Nr : l'en-cours minimal doit permettre d'éviter la rupture d'approvisionnement du poste aval, dans le cas de la gestion des lectures, cette taille limite dépend du nombre maximum d'opérandes pour chaque type d'opération allouée lors de l'ordonnancement des opérations de l'unité de traitement.

  Nr_L = nombre de lectures parallèles max

  Nr_E = nombre d'écritures parallèles max

A partir de ces différents paramètres, il faut définir comment gérer correctement les lectures et les écritures des données en fonction des opérations à exécuter. L'exemple de la section suivante illustre l'influence des paramètres sur l'ordonnancement des opérations de l'unité de traitement

V.3.1. Exemple pédagogique

Nous considérons le graphe flot de signaux de la Figure V-5. Nous supposons que *x(0)* est l'échantillon d'entrée et que *x(1), x(2) et x(3)* sont des échantillons provenant de l'unité de mémorisation. Les échantillons *x(1), x(2) et x(3)* sont vieillis entre deux itérations de la spécification algorithmique. Nous supposons également que les coefficients *h(0), h(1), h(2)* et *h(3)* ont été distribués dans une seule mémoire, différente de celle des échantillons. La cadence d'arrivée de l'échantillon *x(0)* est fixée à 90 ns. Les opérations ont un temps de traversée de 10 ns. Les mémoires utilisées pour stocker les échantillons et les coefficients sont des mémoires SRAM simple port, les lectures et les écritures s'effectuent en 10 ns.

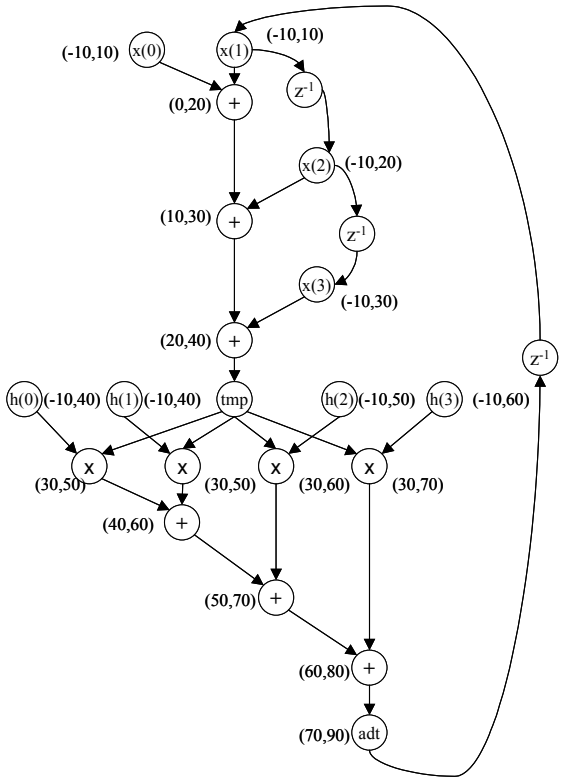


Figure V-5 : Graphe flot de signaux, exemple pédagogique

Les nœuds du graphe flot de signaux sont pondérés par les valeurs correspondant à leur dates *ASAP* et *ALAP* notées (ASAP, ALAP). La date ASAP de la première opération est fixée à 0, les accès en lectures des opérandes se feront en ASAP – T-accès_mem, c'est à dire en –10.

On considère l'ordonnancement basé sur la précédence des opérations et la disponibilité des données. La contrainte de cadence est de 90 ns, le nombre d'opérateurs sélectionnés est, pour cet exemple, d'un multiplieur et d'un additionneur (nombre moyen d'opérateurs). Ceci conduit à l'ordonnancement de la Figure V-6.

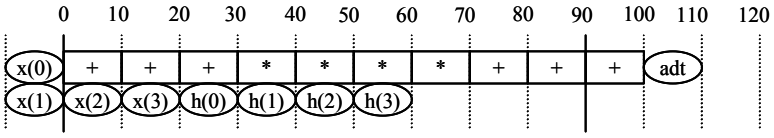


Figure V-6 : ordonnancement avec l'approche classique de la gestion mémoire

Pour satisfaire la cadence de 90 ns, il faut augmenter le nombre d'opérateurs et exploiter le parallélisme de calcul de l'application. Seul le nombre de multiplieurs pourra être augmenté ici car les opérations d'addition sont toutes dépendantes et donc séquentielles. En sélectionnant un plus grand nombre de multiplieurs, le problème se situe non plus au niveau de parallélisme de traitement mais au niveau du parallélisme des accès aux données *h(i)* et *x(i)*. Il faut au minimum deux accès en parallèle ce qui implique une révision de la distribution des données en mémoire. **Pour satisfaire les contraintes temporelles tout en conservant la distribution mémoire prédéfinie, il est nécessaire d'anticiper les lectures des coefficients *h(0), h(1), h(2) et h(3)* par la mise en œuvre d'une gestion par kanban**. Pour obtenir un résultat de synthèse satisfaisant les contraintes temporelles et garantissant un coût minimum, il faut dimensionner le nombre de registres. Dans cet exemple pédagogique nous présentons différents choix de dimensionnement. Pour chacun d'entre eux, nous dégageons des paramètres nous permettant de mettre en œuvre une politique de gestion des accès par kanban à base de files.

Premier choix : nous définissons une taille de file de registres pouvant contenir toutes les données à lire. Pour le *SFG* de l'exemple, seuls quatre échantillons sont lus en mémoire. Le nombre maximum de registres est alors quatre ce qui permet de contenir tous les échantillons en registre et d'effectuer les calculs en parallèle. A chaque cycle, une lecture des coefficients est possible (mémoire SRAM simple port). L'ordonnancement des opérations et des accès mémoire est alors le suivant :

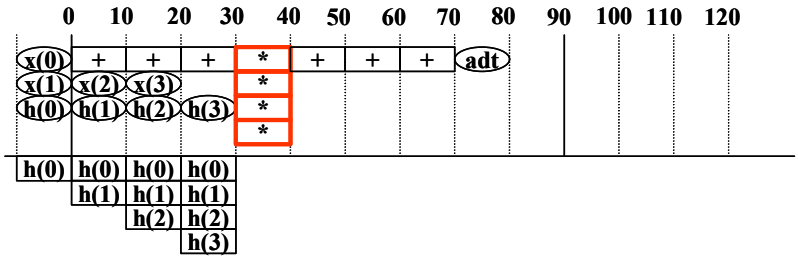


Figure V-7 : Ordonnancement avec une taille kanban de 4

Avec cette taille, nous garantissons la contrainte temporelle tout en conservant la distribution initiale des coefficients dans une seule mémoire. Dans ce cas, l'ordre des lectures n'influence pas l'ordonnancement des opérations puisque les données ont toutes été pré-chargées dans l'unité de traitement avant que l'on puisse ordonner les opérations qui en dépendent. Cependant le surcoût matériel est très élevé puisque pour satisfaire les contraintes temporelles, il faut ajouter 3 multiplieurs et 3 registres. Pour réduire le coût matériel, nous réduisons la taille des files.

Second choix : nous fixons à 2 la taille de la file pour la lecture des coefficients. Dans ce cas, l'ordre de lecture des données va avoir un impact sur le respect ou non de la contrainte temporelle.

Nous examinons deux solutions arbitraires. La première impose que les données soient accédées dans l'ordre suivant : *h(3), h(2), h(1)* puis *h(0)*. La seconde impose l'ordre de lectures *h(0), h(1), h(2)* puis *h(3)*. La Figure V-8 illustre l'influence de l'ordre des lectures sur l'ordonnancement et sur le respect des contraintes.

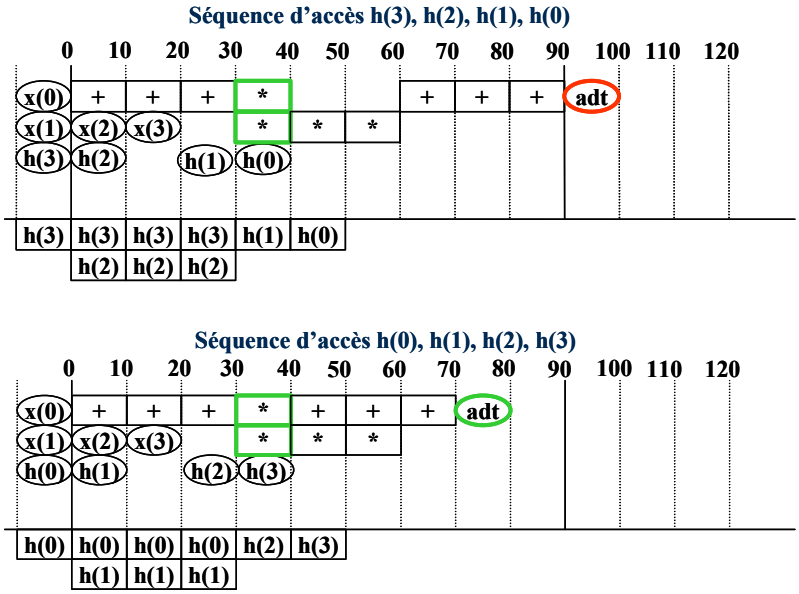


Figure V-8 : Ordonnancement avec une taille kanban de 2 et influence de l'ordre des accès mémoire

Les additions pour le calcul de *adt* ne peuvent s'effectuer qu'à partir de la production du résultat de la multiplication ayant comme opérande *h(0)*. Pour la première solution, on ne peut pas ordonnancer les additions avant la fin des multiplications puisque la dernière addition dépend de *h(0)* ; la contrainte de cadence d'arrivée de l'échantillon *x(0)* ne peut pas être respectée. Pour la seconde, les additions peuvent s'effectuer après la fin de la première multiplication puisque *h(0)* est la première donnée lue, et de ce fait, on respecte la cadence de 90 ns. Cet exemple illustre l'influence de l'ordre des accès aux données : ici les données doivent être lues suivant **l'ordre croissant de leur date ALAP**. L'ordonnancement des anticipations des

lectures doit être cohérent avec l'ordonnancement (list-scheduling) des calculs. Cette cohérence est garantie par le tri des données à lire suivant leurs dates ALAP. Pour optimiser l'ordre d'accès des lectures et écritures, nous utiliserons les ordonnancements ASAP puis ALAP du graphe flot de signaux. Nous fixons statiquement l'ordre des accès en lecture et en écriture en fonction :

- de la distribution des données dans les différentes mémoires allouées.
- de la distribution des données sur les différents types d'opération sélectionnées pour réaliser les calculs de l'unité de traitement.

Nous proposons un formalisme permettant de définir l'ordre des lectures et des écritures de toutes les données placées en mémoire.

Dans un premier temps nous rappellerons comment sont définies et utilisées les files gérant le statut des opérations de l'unité de traitement lors de l'ordonnancement. Puis nous introduirons les nouvelles files pour la gestion kanban des lectures et des écritures. Enfin nous dimensionnerons les files et listes pour gérer le flux d'accès entre unité de traitement et unité de mémorisation.

V.3.2. Définition des files utilisées pour l'ordonnancement des opérations

L'ordonnancement par liste de priorités des opérations de l'unité de traitement est réalisé à l'aide de files dans notre outil de synthèse *GAUT*. Ces files permettent de gérer l'ensemble des opérations à ordonnancer suivant leur type et leur état.

Pour chaque ensemble de type *f* d'opération, trois files à base de *FIFO* sont définies pour différencier les opérations exécutables, les opérations en cours d'exécution et les opérations en cours d'exécution dont les opérateurs pipeline ont libéré leurs premiers étages de pipeline.

- Les files *FileS[f].[EnsX]* contiennent l'ensemble des opérations réalisant la fonction *f* qui sont exécutables à un instant *t* donné. Une opération est dite exécutable lorsque l'ensemble de ses prédécesseurs a été réalisé.
- Les files *FileS[f].[EnsC]* contiennent l'ensemble des opérations réalisant la fonction *f* qui occupent un opérateur à un instant *t* donné.
- Les files *FileS[f].[EnsCC]* contiennent l'ensemble des opérations réalisant la fonction *f* qui occupent un opérateur pipeline à un instant *t* donné dont le premier étage est libre et autorise l'opérateur à accepter une nouvelle opération.

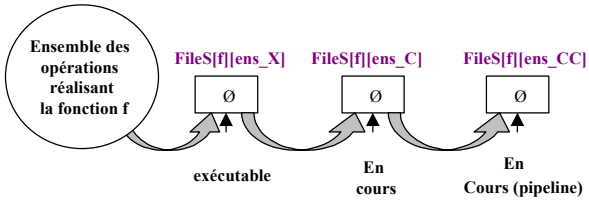


Figure V-9 : Files utilisées pour gérer l'ordonnancement des opérations de l'unité de traitement.

Pour simplifier la lecture, nous nommerons par la suite les files, *FileS[f] [EnsX], FileS[f] [EnsC], FileS[f] [EnsCC]* respectivement *EnsX, EnsC et EnsCC*.

V.3.3. Définitions des nouvelles files et listes

La mise en œuvre de la gestion de type kanban dans l'ordonnancement des opérations amène à la définition de nouvelles files. Ces files servent uniquement à gérer l'ordonnancement ; elles ne sont pas implantées dans l'architecture de l'unité de traitement. Les différentes files et listes utilisées permettent de gérer :

- les lectures des données dans les mémoires allouées.
- les accès aux données qui doivent être utilisées par un type d'opération donné.
- les données qui ont été consommées par les opérations de l'unité de traitement.
- les données qui doivent être écrites dans les différentes mémoires allouées.
- les données qui ont été écrites en mémoire.

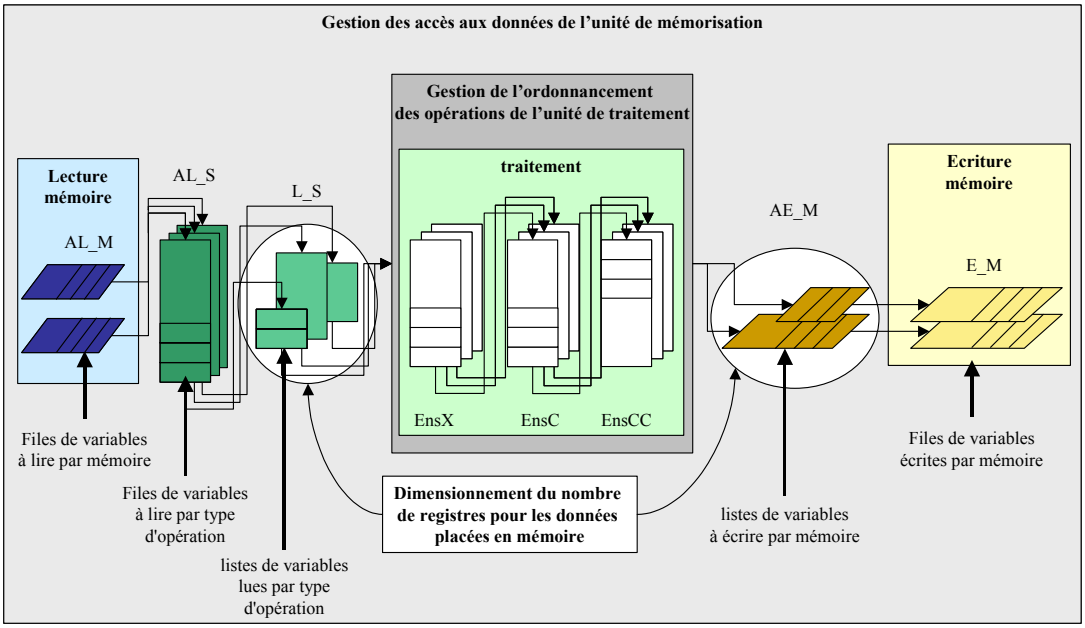


Figure V-10 : Introduction de files pour gestion kanban.

Ces nouvelles files et listes devront permettre d'anticiper les accès mémoire dans le temps et de réduire la complexité de l'unité de mémorisation tout en conservant une complexité de l'unité de traitement raisonnable. La complexité de l'unité de traitement dépend du nombre de registres

nécessaires au stockage des données dont les lectures ont été anticipées. Il faut garantir les contraintes temporelles (augmenter le nombre de registres) et limiter la complexité de l'architecture de l'unité de traitement (réduire le nombre de registres).

La Figure V-10 représente la gestion des différentes files à mettre en œuvre pour la gestion des accès mémoire par kanban. Elle illustre les besoins pour une distribution des données dans deux mémoires et pour l'ordonnancement trois types d'opération *f*.

- Les files permettant de gérer l'ordre des accès en lecture à la mémoire sont appelées files *AL_M*, nous reviendrons sur le classement des données dans la file *AL_M* dans la section V.4.
- Les files, notées *AL_S*, contiennent les données à lire en fonction du type d'opération dont elles sont les opérandes files. Le classement des données dans ces files sera également discuté dans la section V.4.
- Les listes *L_S* contiennent les données qui ont été lues en mémoire suivant le type d'opération à exécuter et dont les opérations n'ont pas été ordonnancées.
- Les listes *AE_M* contiennent tous les résultats fournis par les opérations ordonnancées et qui doivent être placés.
- Les files *E_M* contiennent les données qui ont été produites par l'unité de traitement, les données contenues dans cette file vont être réinjectées dans les files *AL_M* lorsque les données doivent à nouveau être lues dans l'itération courante de l'algorithme.

Nous allons détailler comment les files sont utilisées pour gérer les lectures et les écritures. La gestion des files *L_S* et *AE_M* est une adaptation de la gestion par kanban. L'originalité de l'approche est la gestion de l'orthogonalité entre les lectures en mémoire (parallélisme d'accès) et leur consommation par les opérations de l'unité de traitement (parallélisme de calcul). L'introduction des files *AL_M* et *AL_S* permet de répondre à ce problème lors de l'ordonnancement.

V.4. Gestion des lectures

La méthode de gestion employée pour gérer les accès en lecture par anticipation va permettre de limiter la complexité. L'ordre des accès à la mémoire va être défini statiquement. Les données à transférer vont être classées statiquement à partir des dates ASAP et ALAP du *SFG*. Ce tri réalisé statiquement permet de définir l'ordre dans lequel les données vont être lues. Ceci permet de réduire la complexité de l'ordonnancement puisque le choix des transferts de données n'est plus réalisé dynamiquement. Seule la gestion des variables disponibles dans l'unité de traitement devra être gérée dynamiquement. Les différentes files et listes utilisées pour gérer l'anticipation des lectures vont être détaillées ci dessous.

a. Les files *AL_M*.

Les données sont distribuées en mémoire et pour chaque mémoire allouée, nous définissons une file contenant toutes les données qui doivent y être lues. Lors de l'ordonnancement, à chaque cycle d'accès à la mémoire, un nombre de données dépendant de la mémoire sélectionnée (simple ou multiports …), seront défilées et seront considérées disponibles pour l'ordonnancement des opérations de l'unité de traitement. Nous allons détailler comment doivent être classées les données dans les files de façon à obtenir un ordonnancement efficace. Nous réalisons un ordonnancement *ASAP-ALAP* du graphe flot de signaux. Dans le cas des lectures, nous utiliserons les dates *ALAP* des nœuds de données qui doivent être lues pour déterminer leurs positions dans une file. Pour déterminer les dates *ALAP* des nœuds de données accédant à la mémoire, il faut connaître la date *ALAP* de tous ses nœuds successeurs. La date *ALAP* du nœud de données est le minimum des dates *ALAP* de ses successeurs moins le temps d'accès à la mémoire.

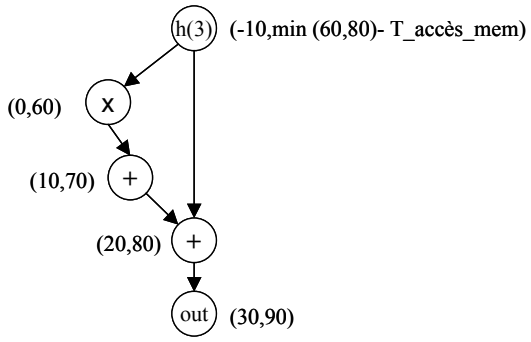


Figure V-11 : Détermination des dates ALAP des nœuds accédés en lecture

En classant les données suivant l'ordre croissant de leur date *ALAP*, nous garantissons que les données placées dans les mémoires seront fournies dans l'ordre le plus adéquat pour l'ordonnancement des opérations.

b. Les files *AL_S*

Les données distribuées en mémoire ne sont pas forcément les opérandes d'une seule opération. Nous définissons des files contenant les opérandes placés en mémoire pour chaque type d'opération *f* sélectionnée. Les données seront là aussi classées suivant une date *ALAP* qui diffère en fonction des opérations. Nous fixerons des dates *ALAP* pour chaque type d'opération utilisée dans le graphe. Ce classement permet de définir une priorité d'accès aux données suivant le type d'opération à réaliser.

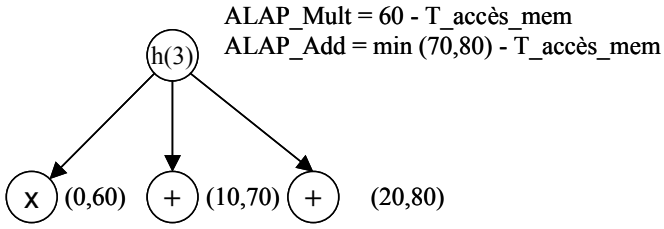


Figure V-12 : Détermination des dates *ALAP* de lecture en fonction des opérations

Les dates ALAP des nœuds en fonction de leur opération sont déterminées de la manière suivante :

$$ALAP(V(f))=Min(ALAP(f))-T\_accès\_mem(V)$$

où V est le nœud de données qui doit être lu en mémoire et f le type de fonction dont V est un opérande.

Les séquences d'accès aux données sont différentes d'une file à l'autre. Nous utiliserons les files *AL_M* pour gérer les lectures car elles fournissent des séquences de données par rapport aux plus petites valeurs des dates *ALAP* de chaque nœud de données. Les files *AL_S* indiquent seulement la priorité d'accès des données par rapport à un type d'opération. Cette file est constituée une fois pour toute avant l'ordonnancement.

c. Les listes *L_S*

Ces listes sont gérées dynamiquement au cours de l'ordonnancement ; elles vont permettre de gérer le flux de données entre l'unité de mémorisation et l'unité de traitement.

Les données lues en mémoire à chaque cycle d'accès sont rangées dans les listes *L_S* en fonction du type d'opération dont elles sont les opérandes. Elles seront classées dans ces listes en fonction des dates *ALAP* déterminées pour chaque type de fonction.

Si les données placées dans une liste *L_S* ne sont pas consommées par le traitement, elles restent dans la liste ; une fois consommées par le traitement elles sont retirées de la liste. Le flux entre l'unité de mémorisation et l'unité de traitement est géré par la taille de la liste. Les lectures en mémoire sont inhibées lorsqu'une liste *L_S* est pleine. Dans ce cas, il faut ordonnancer les opérations dont les opérandes sont contenus dans la liste, avant de pour pouvoir relancer le processus de lecture de données en mémoire. Ceci revient à dimensionner le nombre de kanban qui peuvent être utilisés pour chaque type d'opération. Nous pouvons déterminer les bornes maximales des listes *L_S* en fonction du nombre d'opérandes qui sont potentiellement accessibles en parallèle lors de l'ordonnancement.

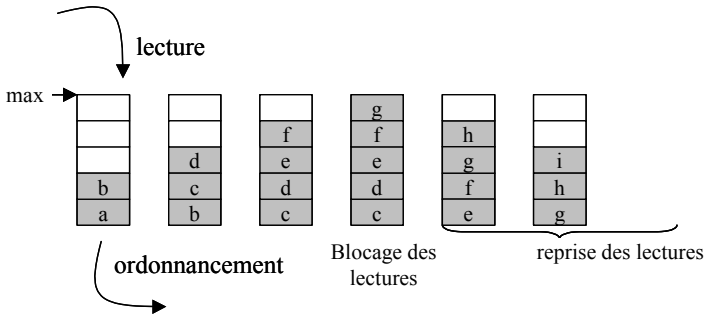


Figure V-13 : Gestion des listes L_S

Le passage des listes "éléments à lire" aux listes "éléments lus" rend les opérations exécutables puisque tous les opérandes d'une opération *f* doivent être dans sa liste *L_S* correspondante avant que cette opération puisse être exécutée. Ceci transforme les règles du critère d'accessibilité

V.5. Gestion du traitement

Pour qu'une opération soit exécutable, il faut qu'elle respecte les deux critères suivants. Toutes les opérations dont elle dépend doivent être exécutées, c'est à dire que le temps de fin de l'ensemble de ses prédécesseurs doit être inférieur au temps courant. De plus les opérandes doivent être accessibles, c'est à dire que l'ensemble des données nécessaires à l'opération est présent soit dans la liste *L_S*, soit en registre pour les données dites locales. Toute opération ne respectant pas ces deux critères ne peut être ordonnancée.

La mise en œuvre ne change pas fondamentalement la gestion de l'ordonnancement des opérations de l'unité de traitement où seul le critère d'accessibilité a changé. La seconde modification vis à vis de l'ordonnancement est qu'il peut être interrompu si le nombre d'écritures en mémoire atteint un seuil critique. Ce point sera détaillé dans le paragraphe suivant.

V.6. Gestions des écritures

En ce qui concerne les écritures en mémoire le poste amont est l'unité de traitement et le poste aval, l'unité de mémorisation. Dans ce cas, nous ne sommes pas obligés de tenir compte du type d'opération qui produit une donnée pour gérer les écritures en mémoire. Nous définissons une file contenant les données produites par tout type d'opération de l'unité de traitement pour chaque mémoire. Ce sont les files de données à écrire dans une mémoire donnée *(liste AE_M)*. Les *files E_M* contiendront les données qui ont été écrites en mémoire.

a. Les listes *AE_M*

Ces files sont gérées dynamiquement ; elles vont permettre de gérer le flux de données entre l'unité de traitement et l'unité de mémorisation.

Lorsqu'une opération se termine (les opérations sont défilées des ensembles *EnsC* pour les opérations réalisées sur un opérateur non-pipeline et de *EnsCC* pour les opérations réalisées sur un opérateur pipeline), les variables produites peuvent être mises en mémoire. Les données qui doivent être écrites en mémoire sont placées dans les listes *AE_M* en fonction de leur mémoire de destination. Elles seront classées dans ces files en fonction des dates *ASAP* des opérations qui les ont produites et des temps de traversée des opérations. Le graphe flot de signaux partiel de la Figure V-14 illustre le calcul des dates *ASAP* des nœuds de données devant être écrites en mémoire.

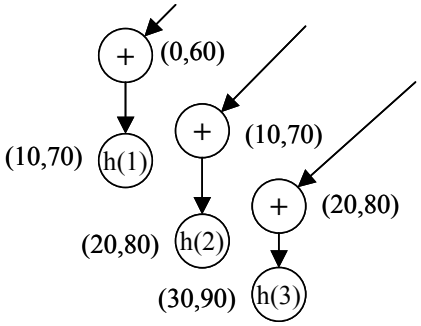


Figure V-14 : détermination des dates *ASAP* des nœuds de données à écrire en mémoire

Il faut vérifier que les mémoires associées aux files sont accessibles. Si les mémoires sont accessibles, les variables sont défilées des listes *AE_M* et enfilées dans les files *E_M*. Dans le cas contraire, elles restent dans les listes *AE_M*. Pour gérer le flux de données entre unité de traitement et unité de mémorisation, les listes *AE_M* sont bornées. Ceci revient à dimensionner le kanban pour les écritures vers les différentes mémoires. Nous pouvons déterminer les bornes maximales des listes *AE_M* en fonction du nombre d'opérations fournissant une donnée à placer en mémoire qui sont potentiellement réalisables en parallèle lors de l'ordonnancement. Les bornes minimales sont égales à 1.

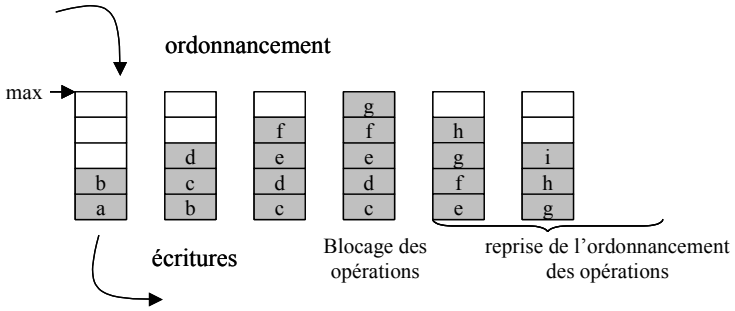


Figure V-15 : Gestion des écritures

Pour un nombre de kanban donné, lorsque la liste *AE_M* est remplie, il faut impérativement écrire en mémoire, les opérations de l'unité de traitement qui produisent des données à écrire en mémoire sont bloquées tant que la liste *AE_M* est pleine. Ce mécanisme permet de réguler les accès mémoire en écriture.

b. Les files E_*M*

Les files *E_M* servent à conserver toutes les données qui ont été écrites en mémoire. Certaines des données placées dans ces files vont être recopiées dans les files *AL_M* et *AL_S* car elles sont à nouveau lues (cf Figure V-16).

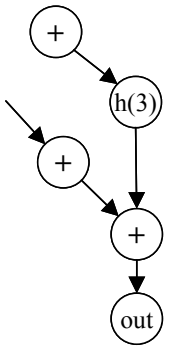


Figure V-16 : Ecriture suivie d'une lecture

Dans ce cas, la donnée *h(3)* ne peut être lue qu'après son passage de la file *AL_M* à la file *L_S* et lorsque la donnée a été placée dans la file *E_M* indépendamment de la date *ALAP* de *h(3)* définie la file *AL_M*. Ceci permet d'assurer la cohérence des données lors de l'ordonnancement.

V.7. Gestion des accès en mode pipeline

Le modèle d'architecture ciblé par la synthèse de haut niveau de *GAUT* est un modèle d'architecture pipeline. La gestion mémoire pour les architectures pipeline pose les trois problèmes suivants :

- Lorsque l'ordonnancement est opéré, il faut qu'à la date *Tcourante*, on vérifie la disponibilité de l'accès mémoire aux différentes dates *T_modulo[cadence]*. En effet les accès se répètent identiquement à chaque itération de l'algorithme ; un conflit d'accès mémoire peut donc intervenir entre deux tranches pipeline différentes.
- Il faut vérifier que l'intervalle de temps (durée de vie) qui sépare l'écriture d'une donnée de sa dernière lecture ne dépasse pas le temps de cadence de l'architecture.
- La notion d'anticipation des lectures par gestion kanban est orthogonale à la réduction des durées de vie des données dans l'unité de traitement. Cependant nous pouvons introduire un mécanisme n'autorisant la lecture des données en mémoire à un instant *t* que lorsque nous garantissons que la donnée sera consommée par l'unité de traitement avant *t* + *cadence*. Cependant, il peut être utile de le mettre en œuvre pour gérer les données rebouclées (ou adaptées) puisqu'une durée entre lecture et écriture supérieure au temps de cadence conduit à un échec de la synthèse. Cette stratégie de contrôle des

accès aux données rebouclées devra être intégrée à notre gestion par kankan des accès mémoire.

V.8. Conclusion

La gestion des lectures de l'ordonnancement des traitements et des écritures est régulée par le flux tiré de l'ordonnancement. La nature du flux tiré permet de coordonner les lectures et les écritures. La gestion par kanban privilégie une écriture en mémoire par rapport à une lecture lorsqu'il y a un conflit d'accès écriture/lecture. L'ordonnancement des opérations produisant des données à écrire en mémoire est piloté par les accès aux différentes mémoires.

Les lectures sont régulées par le traitement des opérations dont les données distribuées en mémoires. Si le mécanisme de gestion des lectures vers l'unité de traitement est une adaptation de la gestion par kanban ; une innovation permettant de prendre en compte la distribution des données en mémoire a été apportée pour gérer l'orthogonalité entre la mémoire et le traitement.

La gestion statique des accès à la mémoire permet de réduire la complexité de l'algorithme d'ordonnancement.

La gestion des accès par kanban permet de conserver une architecture et une distribution des données prédéfinies tout en satisfaisant des contraintes temporelles strictes à un coût moindre : maîtrise du nombre de registres, limitation du nombre de bancs mémoire. Cela apporte un gain de temps au concepteur qui n'est pas obligé de revoir entièrement sa stratégie de mémorisation. Cette gestion permet également un schéma d'accès à l'unité de mémorisation régulier qui permet de contrôler la consommation due aux accès mémoire parallèles. L'originalité et l'efficacité de la méthode de gestion des accès par flux tiré seront développées et validées dans le chapitre suivant.

Chapitre VI
Applications TDSI

Dans ce chapitre nous exposons les résultats de synthèse d'unités de traitement réalisées sous contraintes temporelles. Le choix des applications à synthétiser a été guidé par l'expérience acquise au LESTER dans les domaines du traitement du signal, de l'image et des télécommunications.

Nous débutons ce chapitre par un aperçu de l'environnement utilisateur de l'outil GAUT puis nous produirons plusieurs champs d'expérimentation. Dans la première expérience nous présentons un exemple tutoré de synthèse d'architecture sous contrainte de mémorisation. Le second champ d'expérimentation va mettre en avant l'intérêt des stratégies de la gestion des vecteurs vieillissants développées dans le chapitre IV ainsi que celle de la gestion par flux tiré des accès du chapitre V. Nous explorons l'espace de solutions pour des applications réalisant des annulations d'écho acoustique qui utilisent des données dites vieillissantes et ont un parallélisme de donnée et de traitement.

Nous terminons ce chapitre par deux expériences, la première montre l'intérêt de la modélisation des contraintes de mémorisation et des mécanismes de gestion des accès. La seconde illustre la capacité de notre approche et de notre outil de synthèse d'architecture à traiter des problèmes industriels de grande complexité.

VI.1. L'outil *GAUT*.

Dans ce chapitre, nous présentons comment réaliser une synthèse sous contraintes mémoire à l'aide de l'outil *GAUT*.

Nous présentons d'abord l'interface graphique puis nous détaillerons les différentes phases permettant de réaliser une synthèse sous contraintes de mémorisation. La Figure VI.1 présente l'interface de l'outil.

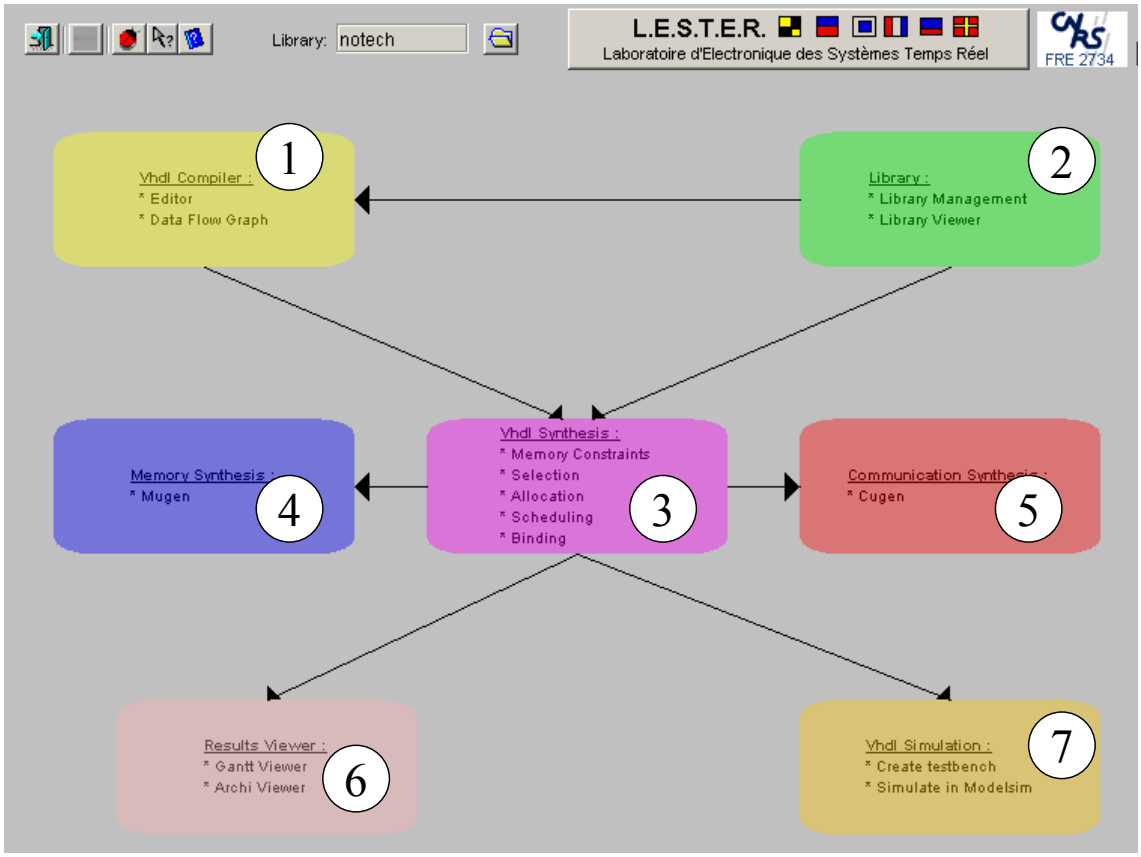


Figure VI.1 : Interface de l'outil *GAUT* : les 7 étapes de la synthèse

-1- Cette étape permet l'édition de spécification en VHDL comportemental. La spécification est compilée et parallélisée pour extraire le graphe flot de signaux. Le graphe généré peut être visualisé.

-2- Elle permet de sélectionner les bibliothèques caractérisant les cibles technologiques visées.

-3- L'étape trois est le cœur de la synthèse. Elle permet de spécifier les contraintes des entrées/sorties de la spécification, les contraintes mémoire et les contraintes de cadence. C'est dans cette fenêtre que la synthèse de l'unité de traitement est lancée. Les phases de sélection, d'allocation, d'ordonnancement et d'assignation sont réalisées et l'architecture de niveau RTL de l'unité de traitement est générée.

-4- Cette étape permet de générer l'architecture de l'unité de mémorisation.

-5- Elle permet de générer l'interface de communication.

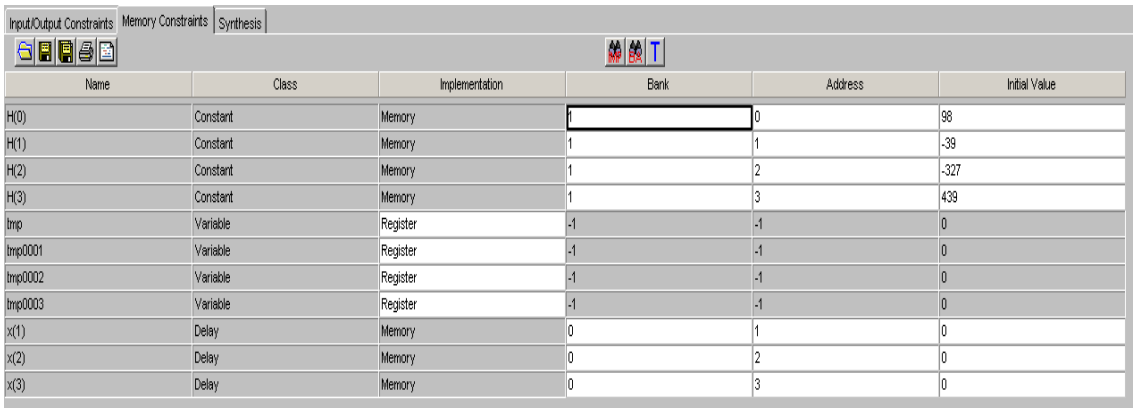
-6- L'étape 6 permet de visualiser les diagrammes de Gantt de l'unité de traitement, des accès aux entrées/sortie et des accès aux mémoires.

-7- Enfin, le composant généré peut être simulé via une interface avec l'outil de simulation ModelSim.

Nous allons présenter les étapes 3, 4 et 6 sur un exemple simple (FIR 4 points) qui permettra de montrer comment réaliser une synthèse intégrant les contraintes mémoires.

Etape 3 : Synthèse sous contraintes mémoire.

L'utilisateur peut, une fois le graphe flot généré, définir une distribution et un placement des données en mémoire. Dans le cas du Fir 4 points nous plaçons par exemple les coefficients *X* dans le banc mémoire 0 et les échantillons *H* dans le banc mémoire 1.



Name	Class	Implementation	Bank	Address	Initial Value
H(0)	Constant	Memory	1	0	98
H(1)	Constant	Memory	1	1	-39
H(2)	Constant	Memory	1	2	-327
H(3)	Constant	Memory	1	3	439
tmp	Variable	Register	-1	-1	0
tmp0001	Variable	Register	-1	-1	0
tmp0002	Variable	Register	-1	-1	0
tmp0003	Variable	Register	-1	-1	0
x(1)	Delay	Memory	0	1	0
x(2)	Delay	Memory	0	2	0
x(3)	Delay	Memory	0	3	0

Figure VI.2 : Interface permettant de définir une distribution et un placement en mémoire

Pour effectuer une synthèse, le concepteur doit choisir une cadence, la fréquence de l'horloge de l'architecture et les contraintes qu'il veut appliquer (contraintes mémoire ou contraintes de communication). Il peut également choisir d'effectuer un certain nombre d'optimisations (optimisation du nombre de registres de l'unité de traitement, optimisation du nombre de bus et optimisation de l'ordonnancement par gestion kanban). Puis il peut lancer le processus de synthèse. S'il a choisi d'effectuer une synthèse avec optimisation de l'ordonnancement par la gestion kanban, le concepteur est invité à dimensionner le nombre de registre par type d'opération sélectionnée pour gérer les accès aux données. Lorsque la synthèse de l'unité de traitement est terminée, le concepteur peut effectuer la synthèse de l'unité de mémorisation.

Etape 4 : Synthèse de l'unité de mémorisation.

La synthèse de l'unité de mémorisation est réalisée automatiquement. Pour l'instant, elle permet la génération d'unité de mémorisation pour des architectures non pipeline et dont les

données des vecteurs vieillissants sont distribuées dans un seul banc mémoire. L'automatisation de la génération des unités de mémorisation supportant les architectures pipeline et la distribution de données de vecteurs vieillissants est actuellement en cours et sera prochainement intégrée à l'outil.

Etape 6 : Visualisation des accès mémoire.

Cette étape permet de vérifier que l'ordonnancement n'a pas créé de conflits d'accès et que les contraintes mémoire ont été respectées lors de la synthèse. Elle permet également, par l'analyse du diagramme de Gantt des accès à l'unité de mémorisation, de réviser les contraintes de placement et de distribution afin d'améliorer les performances des architectures générées. La Figure VI.3 représente le diagramme de Gantt des accès mémoire pour le filtre Fir 4 points à deux bancs mémoire.

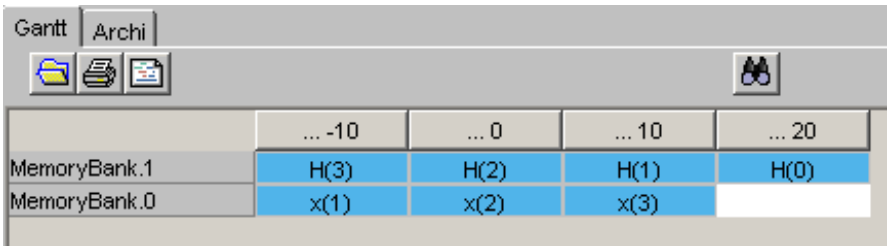


Figure VI.3 : Diagramme de Gantt des accès mémoire

Les différentes étapes que nous avons rapidement parcourues dans ce paragraphe permettent à l'utilisateur de l'outil *GAUT* de spécifier des contraintes mémoire et de réaliser des synthèses intégrant ces contraintes.

VI.2. Exemple de synthèse sous contraintes mémoire

Dans ce chapitre, nous allons décrire le processus de synthèse d'une architecture pour un exemple simple. L'objectif est dans un premier temps de montrer l'obtention d'une solution architecturale sans contrainte de mémorisation. Puis nous introduirons la synthèse sous contraintes de mémoire dite classique. Enfin, nous réaliserons la synthèse intégrant la gestion des accès mémoire par kanban. Un bilan permettra de dégager les caractéristiques principales des différentes architectures obtenues. Pour cette première expérience nous allons réaliser la synthèse d'un filtre adaptatif lms 8 points. Le détail de l'application sera présenté dans la section VI.3. le but étant d'illustrer les différentes synthèses possibles vis à vis de la gestion de la mémorisation.

VI.2.1. Synthèse sans contrainte mémoire

Dans ce premier exemple nous allons détailler tout le processus de synthèse. Le filtre adaptatif "Lms" est décrit en vhdl comportemental.

```
entity lms is
        port (--clk: in bit; xt,yt:in integer; yft:out integer);
end lms;
architecture behavioral of lms is
        constant l : integer := 8;
        constant latency : time := 31250 ns;
        type vecteur is array (0 to l) of integer;
    begin
        lms8 : process
                variable x, h: vecteur;
                variable adapt, deux_mu, y, i:integer;
        begin
                adapt := deux_mu * (yt - y);
                -- adaptation des coefficients du filtre
                for i in 0 to l-1 loop
                        h(i) := h(i) + adapt * x(i);
                end loop;
                -- filtrage
                y := xt * h(0);
                for i in 1 to l-1 loop
                        y :=  y + x(i) * h(i);
                end loop;
                yft <= y;
                -- reccurency of the algorithm
                for i in l-1 downto 2 loop
                        x(i) := x(i-1);
                end loop;
                x(1) := xt;
                wait for latency;
        end process;
end behavioral;
```

Figure VI.4 : VHDL comportemental du filtre lms 8 points

L'algorithme calcule d'abord la valeur de la sortie et une variable d'adaptation. Les coefficients du filtre sont calculés pour l'itération suivante ; enfin les échantillons sont vieillis, ce qui correspond au glissement de la fenêtre temporelle. Cet algorithme est compilé et parallélisé pour extraire le graphe flot de signaux qui va être utilisé pour effectuer les différentes phases de synthèse.

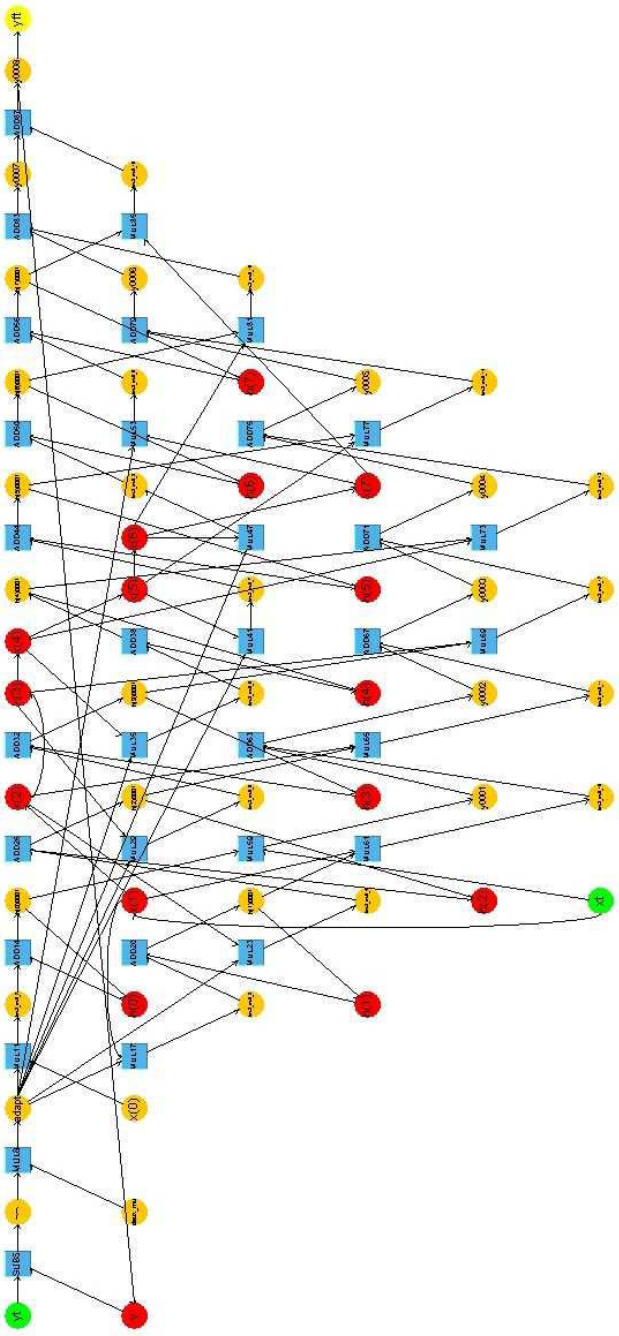


Figure VI.5 : SFG lms 8 points

Nous fixons deux contraintes de cadence : une première à 500ns et une seconde à 300 ns. Les temps de traversée des opérateurs caractérisés dans la bibliothèque sont de 10 ns pour l'additionneur et le soustracteur et de 20 ns pour le multiplieur. La période d'horloge de l'unité de traitement est fixée à 10 ns. Nous effectuons la synthèse avec ces paramètres.

Pour une contrainte de cadence de 500 ns, nous obtenons une architecture non pipeline avec un multiplieur, un additionneur et un soustracteur. Pour une cadence de 300 ns il faut ajouter un multiplieur pour conserver un temps de latence inférieur au temps de cadence.

En sortie de synthèse nous obtenons :

- une description de niveau RTL de l'unité de traitement (lms.vhd) et une description de l'unité de mémorisation (lms_mu.vhd)
- un fichier contenant les informations sur les accès mémoire (lms.mem)
- un rapport contenant les informations sur l'architecture de l'unité de traitement (lms.out)
- des fichiers représentant l'ordonnancement des opérations de l'unité de traitement, les accès mémoire et les entrées sortie sous la forme de diagramme de Gantt, (lms.gantt, lms_mem.gantt, lms_ES.gantt)

Les différents fichiers décrivant l'architecture des unités de traitement et de mémorisation au niveau RTL et les compte rendu de synthèse de cet exemple sont fournis en annexe B.

Les résultats des différentes synthèses sont analysés dans le paragraphe VI.2.3. , Tableau VI-1.

VI.2.2. Synthèse sous contraintes

a. Synthèse classique

Nous appellerons synthèse classique les synthèses réalisées en prenant en compte les contraintes de précédence des opérations et les contraintes de disponibilités des données. Pour cette expérience nous définissons une distribution mémoire où les échantillons sont distribués dans un banc mémoire et les coefficients dans un second. Les mémoires utilisées sont des mémoires SRAM intégrées dans les FPGA de la famille Virtex de Xilinx. Le temps d'accès des lectures et des écritures est de 10 ns. La Figure VI.6 illustre les paramètres nécessaires à la caractérisation d'une mémoire SRAM. La caractérisation de la mémoire inclut le temps d'accès (*time*) en ns, la surface (*area*) en nombre de CLB, la consommation dynamique par accès (*consovar*) en mW/Mhz, la consommation statique (*consostat*) en mW et la taille (*size*) en nombre de bits. Les signaux de contrôle et les ports de lecture/écriture sont également déclarés et dimensionnés.

```
component ram_16x1024
generic
(
      time : integer := 10 ;
      area integer := 4265;
      consovar : integer := 348;
      consocst : integer := 36;
      size : integer := 16384
);
port
(
      a :in td_logic_vector(9 downto 0);
      clk :in std_logic;
      we :in std_logic;
      oe :in std_logic;
      in :in std_logic_vector(nb_bit-1 downto 0);
      out :out std_logic_vector(nb_bit-1 downto 0)
)
end component;
```

Figure VI.6 : Caractérisation d'une mémoire SRAM en bibliothèque

La distribution des données en mémoire s'effectue en utilisant l'interface graphique de l'outil, où le concepteur peut définir la distribution des données de l'algorithme dans différents bancs mémoire. Nous réalisons les synthèses avec les même contraintes temporelles que précédemment.

b. Synthèse avec gestion kanban

Si l'on choisit d'effectuer la synthèse de cet algorithme avec un mode de gestion des accès mémoire par kanban, il suffit de sélectionner l'option de synthèse kanban. La synthèse s'effectue avec la même distribution de données. Le concepteur est invité à dimensionner les kanbans pour chaque type d'opération alloué. Nous utilisons dans cette expérience une taille de 2 et 4 pour les files gérant les données les multiplications correspondant respectivement aux 1 et 2 multiplieurs sélectionnés pour les contraintes de 500 et 300 ns. La taille des files pour les données des opérations addition et soustraction est de 1.

VI.2.3. Comparaison des différentes synthèses

a. Contrainte de cadence de 500 ns

Nous présentons les résultats de synthèse sans contrainte, sous contraintes mémoire et avec gestion par kanban. Le tableau ci-dessous répertorie les principales caractéristiques des différentes synthèses pour l'unité de traitement et l'unité de mémorisation.

		Sans contrainte	Contraintes mémoire	kanban
Unité de traitement	Nbre multiplieurs	1	1	1
	Nbre additionneurs	1	1	1
	Nbre soustracteurs	1	1	1
	Nbre registres	11	12	14
	Latence obtenue	390	400	400
Unité de mémorisation	Nbre de bus	2	2	2
	Nbre d'accès //	2	2	2
	Nbre lectures	32	32	17
	Nbre écritures	9	9	9

Tableau VI-1 : Synthèse pour une cadence de 500 ns

Avec des contraintes relâchées (temps de cadence plus grand), les architectures résultantes des différentes synthèses sont quasiment identiques. Les unités de traitement sont identiques, seul le nombre de registres diffère car les lectures peuvent être anticipées et les données doivent être conservées plus longtemps en registre dans l'unité de traitement.

b. Contrainte de cadence de 300 ns

Les résultats des synthèses montrent, dans ce cas, les limites des synthèses sans contrainte mémoire et avec une gestion classique des accès par rapport à la synthèse intégrant la gestion par kanban. En effet, les résultats du Tableau VI-2 montrent que :

-1- La synthèse sans contrainte de mémorisation conduit à mettre en œuvre une unité de mémorisation plus complexe, avec un nombre de bancs mémoire de 4. Elle sera plus coûteuse en terme de consommation et de nombre d'accès mémoire. Les données qui sont des opérandes de plusieurs opérations sont lues avant chaque exécution d'opération. Dans le cas de la gestion du kanban, une donnée lue reste en registre tant qu'elle n'a pas été consommée par toutes les opérations, ce qui réduit le nombre d'accès à la mémoire.

-2- A cause des contraintes temporelles et de la distribution des données en mémoire, la synthèse classique sous contraintes mémoire ne peut aboutir. Les contraintes d'accès aux mémoires retardent les opérations du traitement. L'architecture devient pipeline et les données récursives sont produites trop tard (cf paragraphe III.2.2.). Les lectures de l'itération $i+1$ s'effectuent avant les écritures de l'itération i ; la synthèse est alors impossible.

		Sans contrainte	Contraintes mémoire	kanban
Unité de traitement	Nbre multiplieurs	2	Synthèse impossible	2
	Nbre additionneurs	1		1
	Nbre soustracteurs	1		1
	Nbre registres	15		19
	Latence	250		250
Unité de mémorisation	Nbre de bus	4	Synthèse impossible	2
	Nbre d'accès //	4		2
	Nbre lectures	32		16
	Nbre écritures	9		9

Tableau VI-2 : Synthèse pour une cadence de 300 ns

Dans ce cas, la gestion par kanban permet non seulement l'obtention d'une solution architecturale pour l'unité de mémorisation à deux bancs mémoire, mais aussi une réduction du nombre d'accès à la mémoire de 50% au prix d'une augmentation de 26% du nombre de registres internes à l'unité de traitement. Sans contrainte, les 8 échantillons et les 8 coefficients sont lus à chaque fois qu'ils sont les opérandes d'une opération. Il faut accéder deux fois à l'ensemble des échantillons et des coefficients pour réaliser toutes les opérations de la spécification dont ils sont les opérandes. Dans le cas de la gestion par kanban, les 8 échantillons et coefficients sont simplement chargés une fois, puis consommés par les opérations de l'unité de traitement. Le nombre de registres de l'unité de traitement augmente légèrement avec la gestion kanban puisque les données dont les lectures ont été anticipées sont conservées en registre.

VI.3. Annulation d'écho acoustique

VI.3.1. Présentation

L'apparition d'écho est un phénomène bien connu dans le domaine de la télécommunication longue distance. Il est, dans ce contexte, introduit par un des composants des récepteurs téléphoniques, qui réinjecte le signal reçu dans le réseau vers l'émetteur du signal. Un nouveau problème d'écho est apparu avec l'avènement du téléphone portable et de la fonction "main libre". L'écho acoustique est, dans ce cas, dû à la réverbération des signaux dans l'environnement de l'utilisateur (voir Figure VI.7).

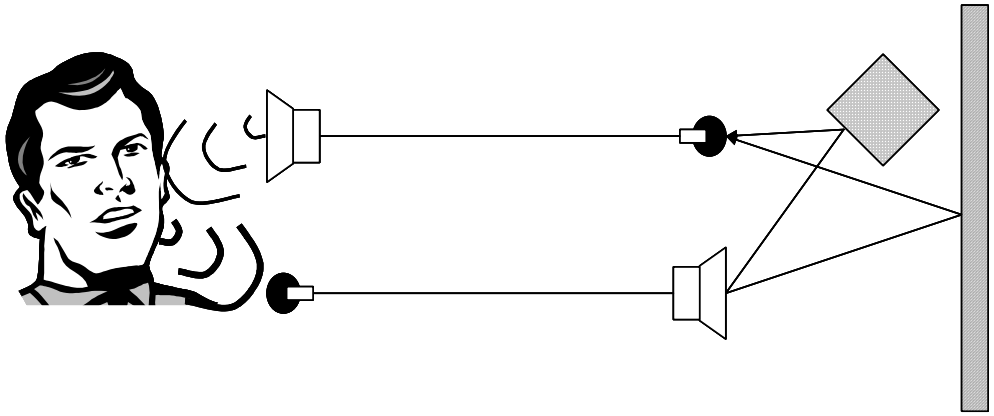


Figure VI.7 : Génération d'écho acoustique

L'acoustique de quasiment tous les locaux est faite de réflexions dues aux murs et aux objets qui y sont disposés. Les échos avec un court délai, de l'ordre d'une dizaine de millisecondes aller-retour, sont perçus comme une distorsion spectrale ou une réverbération. Des échos distincts sont perçus lorsque le délai aller-retour de la réflexion non atténuée est de l'ordre de quelques dizaine de millisecondes. Lorsque le délai s'approche du quart de seconde, l'écho n'est pas atténué et toute conversation devient difficile. Plus le délai est important, plus il faut atténuer le signal réfléchi pour garantir un bon confort d'écoute à l'utilisateur.

Il est alors indispensable de mettre en œuvre des stratégies d'annulation d'écho acoustique par une estimation du canal acoustique de l'écho. La réponse impulsionnelle de la boucle acoustique qui doit être retranchée au signal émis. La variation de la réponse au cours du temps (déplacement de l'utilisateur dans une pièce) impose la mise en œuvre d'un filtrage adaptatif.

VI.3.2. Algorithmes d'annulation d'écho acoustique

Il existe deux grandes familles d'algorithmes d'annulation d'écho acoustique : les algorithmes de moindres carrés récursifs et les algorithmes utilisant le gradient stochastique.

Le principe de l'annulation d'écho par filtrage adaptatif est illustré par la Figure VI.8. Le signal x(t) provenant du haut-parleur est filtré par un système linéaire(chemin d'écho) produisant le signal d'écho y(t) correspondant au signal à émettre. L'algorithme estime la réponse impulsionnelle C_{Li} du chemin d'écho en minimisant l'énergie du signal d'erreur e(t) = y(t)-^y(t).

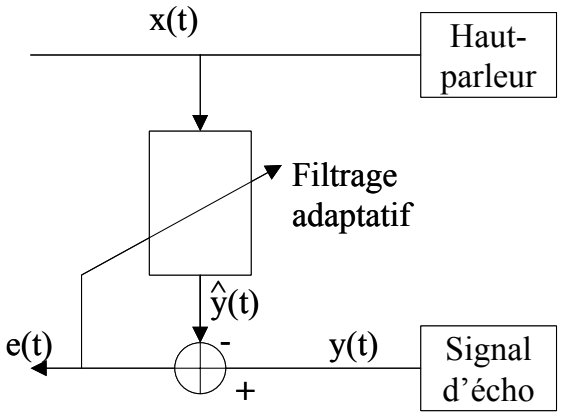


Figure VI.8 : Annulation d'écho par filtrage adaptatif

Il faut identifier le système linéaire en temps réel et traiter l'annulation à la fréquence d'échantillonnage de la parole soit 8 KHz. Il existe deux familles d'identification du canal acoustique :

- les algorithmes de moindres carrés récursifs qui sont optimaux au sens de la convergence mais qui ont une complexité élevée.
- les algorithmes du type gradient stochastique qui possèdent une complexité relativement faible mais des vitesses de convergence médiocres.

Nous allons réaliser dans cette expérimentation une comparaison des unités fonctionnelles de niveau RTL obtenues par synthèse sous contraintes de mémorisation de différents algorithmes de type gradient.

Ces algorithmes décrivent les filtres d'annulation d'écho suivants :

- le LMS (Least Mean Square).
- le NLMS (Normalized Least Mean Square).
- le BLMS (Bloc Least Mean Square) filtre LMS par bloc.

Le filtre LMS est un filtre facile à mettre en œuvre de par sa simplicité ; il permet de mettre à jour les coefficients du filtre adaptatif à l'arrivée d'un nouvel échantillon. L'espérance de l'erreur au carré est approximée par l'erreur au carré elle-même, ce qui simplifie le traitement mais dégrade la convergence surtout pour des signaux à forte dynamique. Pour composer le problème de convergence, on normalise le filtre par rapport à la puissance du signal d'excitation, ainsi nous obtenons un filtre NLMS. Le filtre BLMS détermine les coefficients du filtre non pas à l'arrivée de tout nouvel échantillon, mais seulement tous les K échantillons. Ce filtre permet la

mise en œuvre de techniques de convolution rapide et il est moins sensible au bruit puisque les coefficients restent constants pendant k échantillons (intégration du bruit).

Pour traiter les problèmes d'annulation d'écho acoustique dans une salle de conférence, il faut prendre en compte une bande élargie. La fréquence d'échantillonnage Fe n'est plus de 8KHz mais de 16 voire 32 KHz. La longueur du chemin acoustique porte à une augmentation de l'ordre des filtres adaptatifs à 1024 points. Nous nous plaçons dans ces conditions d'expérimentation. Nous réalisons des synthèses de filtres adaptatifs 1024 points pour des cadences de 62,5 (Fe=16 KHz) et 31,25 µs (Fe=31,25KHz).

a. LMS 1024 points

Synthèse d'architecture non pipeline : nous nous placerons dans des conditions de synthèse telles que l'architecture générée ne soit pas pipeline. Nous réaliserons également des synthèses avec une allocation automatique des opérations. Les résultats de synthèse suivants sont respectivement des résultats pour une distribution du vecteur vieillissant x sur un banc (LMS1b) pour une synthèse classique puis pour la méthode kanban (LMS_kanban_1b).

Cadence 62,5µs		LMS1b	LMS_kanban_1b
Unité de traitement	Nbre multiplieurs	1	1
	Nbre additionneurs	1	1
	Nbre soustracteurs	1	1
	Coût de l'archi (nbre CLB)	141677	150455
	Latence obtenue (ns)	41040	41030
Unité de mémorisation	Nbre de bus	2	2
	Nbre d'accès //	2	2
	Nbre lectures	4096	2048
	Nbre écritures	1047	1047

Tableau VI-3 : Synthèse pour une cadence de 62,5µs (Fe = 16KHz)

Cadence 31,25µs		LMS1b	LMS_kanban_1b
Unité de traitement	Nbre multiplieurs	Synthèse impossible	2
	Nbre additionneurs		1
	Nbre soustracteurs		1
	Coût de l'archi (nbre CLB)		179961
	Latence obtenue (ns)		20580
Unité de mémorisation	Nbre de bus	Synthèse impossible	2
	Nbre d'accès //		2
	Nbre lectures		2048
	Nbre écritures		1047

Tableau VI-4 : Synthèse pour une cadence de 31,25 µs (Fe = 32KHz)

Pour une cadence de 62,5 µs les architectures de synthétisées sont relativement proches. Le surcoût matériel de l'unité de traitement de LMS_kanban_1b est faible (6%). La réduction du nombre d'accès mémoire est importante (40%). Cette réduction du nombre d'accès s'explique par la réduction des durées de vie des variables qui permet de conserver les données en registres internes à l'unité de traitement.

Pour une cadence de 31,25 µs, le nombre de multiplieurs alloués est de 2. Cela implique que 2 échantillons doivent être disponibles pour exploiter au mieux les multiplieurs. Pour la solution LMS_1b, il n'est pas possible de fournir 2 échantillons car ils sont placés dans le même banc mémoire. Le défaut de parallélisme d'accès aux données fait échouer la synthèse. L'introduction du kanban permet d'anticiper la lecture de deux échantillons et d'exploiter les multiplieurs en parallèle. La synthèse est alors rendue possible.

b. NLMS 1024 points

Synthèse d'architecture non pipeline : nous réalisons la synthèse d'un filtre NLMS 1024 points dans les même conditions que l'expérience menée pour le filtre LMS 1024 points.

Cadence 62,5µs		NLMS1b	NLMS_kanban_1b
Unité de traitement	Nbre multiplieurs	1	1
	Nbre additionneurs	1	1
	Nbre soustracteurs	1	1
	Nbre diviseurs	1	1
	Coût de l'archi (nbre CLB)	143523	154099
	Latence obtenue (ns)	41140	41140
Unité de mémorisation	Nbre de bus	2	2
	Nbre d'accès //	2	2
	Nbre lectures	4096	2048
	Nbre écritures	1047	1047

Tableau VI-5 : Synthèse pour une cadence de 62,5µs (Fe = 16KHz)

Cadence 31,25µs		LMS1b	LMS_kanban_1b
Unité de traitement	Nbre multiplieurs	Synthèse impossible	2
	Nbre additionneurs		1
	Nbre soustracteurs		1
	Coût de l'archi (nbre CLB)		182017
	Latence obtenue (ns)		20620
Unité de mémorisation	Nbre de bus	Synthèse impossible	2
	Nbre d'accès //		2
	Nbre lectures		2048
	Nbre écritures		1047

Tableau VI-6 : Synthèse pour une cadence de 31,25 µs (Fe = 32KHz)

La synthèse d'un filtre NLMS 1024 points conduit aux même résultats que pour le filtre LMS 1024 points. L'ordonnancement intégrant la gestion kanban permet d'atteindre des contraintes de cadence plus restrictive sans modifier l'architecture de l'unité de mémorisation.

c. BLMS 8 blocs

Le filtrage est décomposé en 8 blocs de 1024 points, la taille et le nombre d'accès en mémoire vont donc être moins importants.

Cadence 500 µs		BLMS1b	BLMS_kanban_1b
Unité de traitement	Nbre multiplieurs	1	1
	Nbre additionneurs	1	1
	Nbre soustracteurs	1	1
	Nbre diviseur	1	1
	Coût de l'archi (nbre CLB)	49779	54143
	Latence obtenue (ns)	41160	41160
Unité de mémorisation	Nbre de bus	2	2
	Nbre d'accès //	2	2
	Nbre lectures	3136	2112
	Nbre écritures	136	136

Tableau VI-7 : Synthèse pour une cadence de 62,5µs (Fe = 16KHz)

Cadence 250 µs		LMS1b	LMS_kanban_1b
Unité de traitement	Nbre multiplieurs	Synthèse impossible	2
	Nbre additionneurs		1
	Nbre soustracteurs		1
	Coût de l'archi (nbre CLB)		72890
	Latence obtenue (ns)		21880
Unité de mémorisation	Nbre de bus	Synthèse impossible	2
	Nbre d'accès //		2
	Nbre lectures		2112
	Nbre écritures		136

Tableau VI-8 : Synthèse pour une cadence de 31,25 µs (Fe = 32KHz)

Là encore, la synthèse pour une contrainte forte de cadence ne peut être respectée qu'avec l'introduction d'une gestion par kanban. On peut, par ailleurs, distribuer les échantillons sur plusieurs bancs en utilisant l'approche développée dans le chapitre IV. Dans ce cas il faut remettre en cause l'architecture mémoire qui a été prédéfinie pour ces exemples.

VI.4. Filtre de Sobel

VI.4.1. Présentation

Les filtres Prewitt, Sobel, Freeman, et Kirsch portent tous le nom de leurs inventeurs. Ils sont tous conçus dans le même but : détecter avec la plus grande précision les contours naturels "cachés" dans une image CCD. A l'origine ils ont été développés dans le cadre des appareils de vision nocturne, mais ils sont aussi utiles dans l'étude morphologique des objets astronomiques et dans des applications industrielles telles que la détection de pièces ou de contours.

Le filtre de Sobel utilise par exemple deux noyaux 3x3, l'un pour l'axe horizontal (X) et l'autre pour l'axe vertical (Y). Chaque noyau est un filtre gradient. Ils sont combinés pour créer l'image finale.

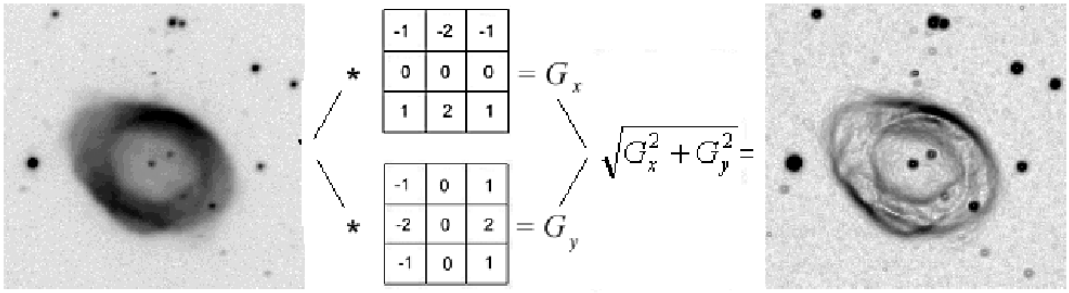


Figure VI.9 : Application d'un filtre de SOBEL à une image

En traitement d'image, l'opérateur linéaire élémentaire pour modifier le contenu de l'image est la convolution, notée *. Mathématiquement, l'opération de convolution correspond, pour chaque pixel de l'image résultante, à l'intégrale (sommation dans le cas discret) du produit du noyau de convolution par la portion d'image au voisinage du pixel d'intérêt.

L'opération de convolution, difficile à visualiser au sens mathématique, peut se ramener dans la plupart des cas à une simple opération de sommation pondérée par les coefficients du noyau, comme il est illustré ci-dessous :

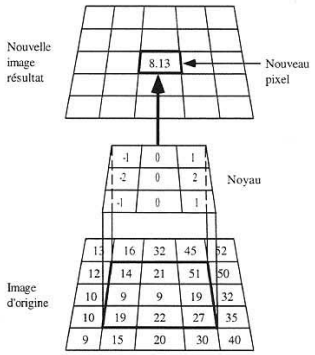


Figure VI.10 : Filtrage d'un pixel

Chaque valeur d'un nouveau pixel sur l'image résultante est calculée en multipliant le noyau de convolution et la section de l'image d'origine centrée au pixel d'intérêt.

$$Nouveau\ pixel = \begin{bmatrix} -1*14+0*21+1*51 \\ -2*9+0*9+2*19 \\ -1*19+0*22+1*27 \end{bmatrix} / 8 = 8.13$$

Figure VI.11 : Calcul d'un pixel

L'opération de convolution se résume donc à la boucle de calcul suivante :

Pour toutes les positions de pixel de la nouvelle image :

- Entrer le noyau à la position du nouveau pixel.
- Les valeurs de pixel sous le noyau sont multipliées par les coefficients correspondants de noyau, qui agissent comme poids de pondération.
- Les résultats de toutes les multiplications sont additionnés ensemble.
- La normalisation de l'image résultat est obtenue en divisant par la somme de la valeur absolue des coefficients du noyau.

Les effets de bord sur le contour de l'image sont minimisés en rajoutant au besoin une rangée ou une colonne à l'image d'origine. La valeur de chaque pixel ajoutée est mise égale à la valeur du pixel voisin (effet miroir).

Le choix du noyau permet d'appliquer divers traitements sur l'image d'origine. Certains noyaux sont utilisés pour filtrer l'image (passe-bas ou passe-haut ou passe-bande) tandis que d'autres sont utilisés pour faire ressortir certaines caractéristiques de l'image. Un noyau de Sobel permet d'extraire les arêtes d'une image. Ce noyau réalise en fait un calcul de dérivée première selon une direction donnée dans l'image. Le noyau SH extrait les arêtes horizontales tandis que le noyau SV extrait les arêtes verticales.

$$SH : \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \qquad SV : \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

Figure VI.12 : Extraction de noyau

Dans les deux cas, le facteur de pondération est 8. La détection d'arêtes de n'importe quelle orientation s'obtient en combinant la sortie des deux noyaux.

$S=\sqrt{SH^2+SV^2}$ valeur exacte, ou

$S=|SH|+|SV|$ valeur approximée, plus rapide à calculer.

VI.4.2. Contexte d'expérimentation

L'objectif est de réaliser un filtre de Sobel sur une image de 512 par 512. Pour réaliser ce filtrage, nous appliquerons un filtre des imagettes 16 par 16. la taille du filtre est dimensionnée à 17 par 17 pour éliminer les effets de bord.

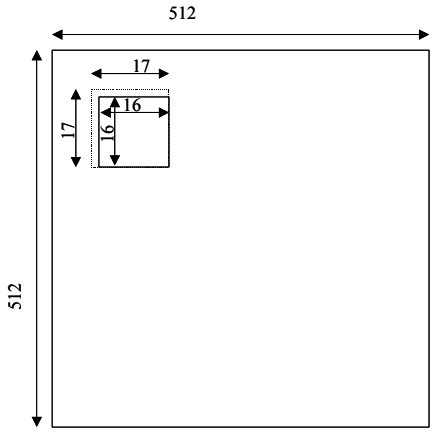


Figure VI.13 : Filtrage d'une image 512 par 512

La latence du traitement d'une l'image 512 par 512 est définie par *Tappli* exprimée en nombre d'images par secondes (Im/s).

$Tappli = Tlat \times n^2$, où *Tlat* est la latence d'un filtrage d'une imagette 16 par 16 et *n* le nombre d'imagettes contenu dans l'image $n = \frac{512}{16}$.

Nous voulons observer le comportement de la synthèse par des contraintes de temps comprises entre 1 et 20 Im/s. La latence du filtre sera alors comprise entre 976 µs et 48 µs. Nous allons réaliser les synthèses pour ces deux contraintes de temps et comparer les architectures de l'unité de traitement et de mémorisation en dressant un bilan temps / coût pour chaque solution.

VI.4.3. Synthèse

La spécification algorithmique du filtre de Sobel représente une centaine de ligne en VHDL. Le graphe flot de signaux généré contient 14163 nœuds et 34561 arcs.

Nous réalisons des synthèses avec des contraintes temporelles fixées à 1 image et 20 images par seconde. La cible technologique visée est un FPGA virtexE de Xilinx ; nous utilisons la bibliothèque caractérisant cette famille lors de la synthèse.

Une première solution consiste à distribuer les arêtes verticales et horizontales dans une mémoire SRAM, les variables temporaires dans une seconde et les constantes dans une mémoire ROM. Cette solution de distribution est conservée tant qu'elle permet de respecter les contraintes temporelles.

Pour la contrainte temporelle la plus forte (20 Im/s), la distribution des données sur 3 mémoires ne permet pas l'obtention d'une architecture. Les données sont distribuées sur 5 mémoires. Les arêtes horizontales sont distribuées dans le banc 1, les arêtes verticales dans le banc 2 et les variables temporaires de calcul *TImTmp* et *TImTmp2* respectivement dans les bancs 2 et 3. Enfin les constantes sont distribuées dans une mémoire ROM. Les mémoires SRAM et

ROM utilisées ont des temps d'accès de 10 ns. Le tableau ci-dessous caractérise les unités de traitement générées pour les deux contraintes temporelles.

	Latence obtenue	Coût (Nbre CLB)
Sobel_3b	54170	518867
Sobel_5b	46670	414507
Sobel_3b_kanban	30870	589325

Tableau VI-9 : Unités de traitement

Solution à 3 mémoires	SRAM1	SRAM2	ROM3
Taille	648 x 16	648 x 16	4
Nombre d'accès	2592	3688	2338
Nombre d'accès kanban	1296	1296	6
Nombre de bit du générateur d'@	10	10	2

Tableau VI-10 : Unité de mémorisation à 3 bancs mémoire

Solution à 5 mémoires	SRAM 1	SRAM2	SRAM3	SRAM4	ROM3
Taille	324x16	324 x 16	324x16	324x 16	4
Nombre d'accès	1296	1296	1844	3688	2338
Nombre de bit du générateur d'@	9	9	9	9	2

Tableau VI-11 : Unité de mémorisation à 5 bancs mémoire

L'architecture de l'unité de traitement générée pour la solution basée sur la gestion des accès par kanban est plus coûteuse (+13% par rapport à une solution 3 bancs et + 29% par rapport à la solution 5 bancs dite classique). Cependant elle offre deux avantages : le premier est qu'elle permet de satisfaire les contraintes temporelles de l'application sans modification de l'architecture de l'unité de mémorisation ; le second est la réduction du nombre d'accès à la mémoire de l'ordre de 50 à 66 % pour les accès aux mémoires SRAM et de plus de 90 % pour les accès à la mémoire ROM. Par ailleurs, la solution architecturale de l'unité de mémorisation à 5 bancs mémoire permettant de respecter la contrainte de 20 images par secondes est beaucoup plus coûteuse car il faut mettre en œuvre 4 générateurs d'adresses sur 9 bits par rapport aux 2 générateurs sur 10 bits de l'architecture mémoire obtenue par gestion kanban.

VI.5. Filtrage et synchronisation (Thales Communications)

La synthèse de l'algorithme de filtrage et de synchronisation est strictement confidentielle. Nous ne décrirons absolument rien de technique à son sujet. Par contre, les résultats sont extrêmement intéressants, dans la mesure où seule Thales Communications a été en mesure de synthétiser avec GAUT, puis de synthétiser physiquement pour l'EPXA, un circuit numérique fonctionnant à 70 MHz alors que la cible de synthèse était de 100 MHz en utilisant les blocs mémoires matériels. Les autres sociétés ont validé par simulation leurs résultats mais, à l'heure de la rédaction, la synthèse physique n'est pas terminée, ni d'ailleurs réclamée pour terminer le projet. L'écart entre les 100 MHz attendus et les 70 MHz obtenus s'explique par le fait que la caractérisation des opérateurs arithmétiques employés par GAUT est un peu trop optimiste. Elle sera refaite dans des conditions plus conformes à l'architecture des cellules de bases des circuits générés. En particulier les temps de traversée des multiplexeurs et des démultiplexeurs seront ajoutés. Néanmoins, cette fréquence de 70 MHz est très largement supérieure aux 40 MHz que les ingénieurs de Thales ont obtenu en développant le circuit de référence en RTL « à la main ».

La conception de ce bloc illustre, grâce à l'emploi de l'outil GAUT et l'intégration de contraintes de mémorisation, la validité et les opportunités d'optimisation offertes dans la conception d'application industrielles.

VI.6. Conclusion

Les méthodes de gestion des accès mémoires intégrées à la synthèse d'architecture permettent par une approche simple, l'obtention d'architectures fonctionnelles d'unités de traitement et de mémorisation pour des applications de TDSI.

Les caractéristiques des différentes applications en TDSI traitées permettent de mettre en avant l'efficacité de notre approche par rapport à une conception manuelle et illustre l'aspect novateur de la gestion mémoire.

Les approches de gestion des vecteurs vieillissants et de régulation du flux d'accès par kanban permettent :

- d'obtenir des architectures mémoire à distribution multi-bancs valides pour les vecteurs vieillissants
- d'améliorer les performances en terme de nombre d'accès mémoire
- de respecter des contraintes temporelles
- de maîtriser le compromis registres internes et mémoire

Les différentes expériences montrent que les solutions de gestion de la mémorisation intégrées dans l'outil de synthèses sont valides et permettent d'obtenir des performances intéressantes pour des applications de complexité industrielle.

Chapitre VII
Conclusion et perspectives

VII.1. Conclusion

Nous avons proposé dans ce mémoire une approche d'intégration des unités de mémorisation dans le flot de synthèse d'architecture pour des applications orientées traitement du signal, de l'image et des télécommunications. Notre approche est basée sur l'utilisation de techniques de synthèse de haut niveau sous contraintes. Les unités fonctionnelles constituant l'architecture cible du composant sont conçues en fonction :

- de la spécificité de l'application
- des caractéristiques génériques de l'architecture l'unité de mémorisation
- des modèles de représentations des contraintes.

Dans ce contexte, la spécification d'application de traitement de l'image et du signal est modélisée par un Graphe Flot de Signaux *(SFG)*. Les caractéristiques technologiques des mémoires, leur sélection, la distribution des données et la cadence d'itération permettent la génération de contraintes de mémorisation. Nous avons développé une analyse formelle des contraintes mémoire qui repose sur des modèles de graphes et sur leurs projections sur une architecture mémoire générique.

Les contraintes d'intégration spécifiées pour chaque mémoire de l'unité de traitement ont d'abord été modélisées par un graphe de contraintes mémoire ciblant une architecture mémoire simple. Ce modèle supporte les contraintes technologiques des mémoires (type de mémoires utilisées, nombre de ports de lecture et d'écriture) ainsi que les contraintes de distribution des données. Le parcours des graphes de contraintes mémoire et des graphes flot de signaux permettent l'intégration de contrainte de mémorisation lors de la synthèse d'architecture. Les architectures des unités de traitement et de mémorisation obtenues sont correctes d'un point de vue fonctionnel.

Nous avons apporté deux améliorations majeures à notre première proposition. La première intègre la gestion des données vieillissantes pour une distribution des données sur plusieurs bancs mémoire. La gestion des vecteurs vieillissants dans la synthèse repose sur un modèle de graphe permettant de modéliser les séquences d'accès aux données sur les itérations successives de l'algorithme. Une projection du modèle est ensuite effectuée sur une architecture mémoire générique. La gestion du vieillissement permet la génération d'architectures mémoire génériques fonctionnelles intégrant le parallélisme d'accès aux données d'un vecteur vieillissant et des architectures pipeline.

La seconde amélioration vise à réguler le flot d'accès mémoire en anticipant les besoins avant que l'ordonnancement des opérations de l'unité de traitement ne l'impose. Nous avons développé une méthode de régulation des accès mémoire inspirée de la gestion de production par flux tiré. La gestion des flux d'accès est réalisée par des files suivant le principe kanban. Ce

principe a été adapté de manière à gérer le stock de données lues et de données écrites et à contrôler le nombre de registres utilisés dans l'unité de traitement pour stocker les données lues et à écrire en mémoire. Cette adaptation de gestion par kanban permet de réguler la consommation due aux accès à la mémoire. L'anticipation des besoins permet également de conserver une distribution des données en mémoire et une architecture mémoire simple pour des contraintes temporelles fortes.

Nous avons présenté un ensemble de résultats obtenus en appliquant notre méthode à des algorithmes des domaines du TDSI et des Télécommunications. Nous avons montré l'intérêt et la validité de la mise en œuvre et de l'utilisation dans l'outil *GAUT* du formalisme de spécification des contraintes de mémorisation que nous proposons. Ainsi, la première expérience a été réalisée sur un exemple simple. Elle montre l'intégration des modèles et des contraintes de mémorisation dans le flot de synthèse de l'outil *GAUT*. La deuxième expérience permet d'explorer l'espace de solutions qu'offre notre approche. La dernière expérience démontre l'applicabilité de notre méthode pour la conception d'un composant intégrant une architecture mémoire complexe et les gains pouvant être obtenus sur une architecture obtenue par synthèse pour une application industrielle.

VII.2. Perspectives

La méthodologie proposée conduit à des résultats de bonne qualité sur les exemples traités, plusieurs améliorations et extensions peuvent cependant être apportées.

Ainsi, un premier ensemble de travaux visant la synthèse de l'unité de traitement doit être réalisé dans le cadre de la gestion par kanban. Un premier point concerne le dimensionnement optimal des différentes files permettant un coût minimal en nombre de registres dans l'unité de traitement. Le second point est l'intégration de la gestion par kanban pour des architectures pipeline. Une analyse et une gestion des accès pour garantir la fonctionnalité des architectures pipeline devront être développées.

Le second point à développer est la prise en compte des contraintes de l'unité de mémorisation et de l'unité de communication. Un premier travail consistant à intégrer les travaux réalisés dans cette thèse et ceux développés dans la thèse de Philippe Coussy [Cous03b] a été effectué. La mise en œuvre d'une approche plus formelle doit être développée.

Un troisième ensemble de travaux d'automatisation doit être mené sur la méthode d'analyse et de vérification des contraintes que nous avons présentées. De plus, un travail doit être mené pour extraire automatiquement la spécification des contraintes liées à la mémorisation des

données. Une première approche est par exemple la réalisation d'une première synthèse qui permet de définir des orientations dans le placement des données.

Ces trois points s'intègrent dans le flot de synthèse de haut niveau. Il reste un travail de définition de la distribution et du placement des données en amont de la synthèse d'architecture pour des applications non déterministes. Ce travail pourra s'appuyer sur le développement de modèles formels de représentations des structures de données et la définition de métriques.

Bibliographie

Listes des publications personnelles

Conférences internationales.

[Cous05a] Philippe Coussy, Gwenolé Corre, Pierre Bomel, Eric Senn, Eric Martin," High-level synthesis under I/O Timing and Memory constraints," In *Proceedings of IEEE ISCAS'05 (International Symposium on Circuits And Systems)*, mai 2005

[Cous05b] Philippe Coussy, Gwenolé Corre, Pierre Bomel, Eric Senn, Eric Martin, " A More Efficient and Flexible DSP Design Flow from MATLAB-SIMULINK," In *Proceedings of IEEE ICASSP'05 (International Conference on Acoustic, Speech and Signal Processing)*, Vol. V p. 61-64, mars 2005.

[Corr04a] Gwenolé Corre, Eric Senn, Nathalie Julien, Eric Martin, " High Level Ageing Vectors Management for Data Intensive Applications," In *Proceedings of ICSES (International Conference on Signal and Electronic Systems)*, septembre 2004.

[Corr04b] Gwenolé Corre, Eric Senn, Nathalie Julien, Eric Martin, " Memory Accesses management during High Level Synthesis," In *Proceedings of ACM CODES+ISSS'04 (International Conference on CO-DESign and System Synthesis)* , pages 42-47, septembre 2004.

[Corr04c] Gwenolé Corre, Eric Senn, Nathalie Julien, Eric Martin, " Memory aware HLS and the implementation of ageing vectors," In *Proceedings of EUROMICRO Symposium on Digital System Design*, pages 88-95, août 2004.

[Corr04d] Gwenolé Corre, Eric Senn, Nathalie Julien, Eric Martin, " A Memory Aware behavioral Synthesis Tool for Real-Time VLSI Circuits," In *Proceedings of ACM GLSVLSI (Great Lake Symposium On Very Large Scale Integration)*, pages 82-85, Apr 2004

[Corr04e] Gwenolé Corre, Eric Senn, Nathalie Julien, Eric Martin, " Memory Aware High-Level Synthesis for Embedded Systems," In *Proceedings of IADIS Applied Computing* , pages 499-506, Mar 2004.

[Corr04f] Gwenolé Corre, Eric Senn, Nathalie Julien, Eric Martin, "A Memory Aware High Level Synthesis Tool," In *Proceedings of ISVLSI (International Symposium on Very Large Scale Integration)*, pages 279-280, Feb 2004.

Conférences nationales.

[Corr05a] Gwenolé Corre, Philippe Coussy, Pierre Bomel, Eric Senn, Eric Martin,"Synthèse Comportementale Sous Contraintes de Communication et de Placement Mémoire pour les composants du TDSI,"In *Proceedings of GRETSI'05 (Colloque sur le Traitement du Signal et de l'Image)*, sept 2005.

[Corr05b] Gwenolé Corre, Nathalie Julien, Eric Senn, Eric Martin," Réduction de l'influence du placement mémoire par la synthèse de haut niveau,"In *Proceedings of FTFC'05 (journées d'études Faible Tension Faible Consommation)*, mai 2005.

[Corr03a] Gwenolé Corre, Nathalie Julien, Eric Senn, Eric Martin," Contraintes mémoire et solution architecturale pour applications TDSI," in *Proceedings of GRETSI (Colloque sur le Traitement du Signal et de l'Image)*, sept 2003.

[Corr03b] Gwenolé Corre, Nathalie Julien, Eric Senn, Eric Martin," Ordonnancement sous contraintes de mémorisation : une optimisation efficace des ressources lors de la synthèse d'architecture," In *Proceedings of FTFC (journées d'études Faible Tension Faible Consommation)*, pages 147-152, may 2003.

[Corr02a] Gwenolé Corre, Nathalie Julien, Eric Senn, Eric Martin," Intégration de la synthèse mémoire dans l'outil de synthèse d'architecture GAUT Low Power," In *Proceedings of JFAAA (Journée Francophone sur l'Adéquation Algorithme Architecture)*, décembre 2002.

[Corr02b] Gwenolé Corre, Nathalie Julien, Eric Senn, Eric Martin," Optimisation de la consommation des unités de mémorisation lors de la synthèse d'architecture," In *Proceedings of GDR-CAO*, pages 169-172, mai 2002.

Bibliographie

[Amar95] S.Amarasinghe, J.Anderson, M.Lam, and C.Tseng, "The SUIF compiler for scalable parallel machines," In *Proceedings of the 7th SIAM Conference on Parallel Proceedings for Scientific Computing*, pages 662-667, 1995.

[Angi03] F. Angiolini et.al., "Polynomial-Time Algorithm for On-Chip Scratchpad Memory Partitioning," In *Proceeding on CASES,* pages 318-326, 2003.

[Avis02] O. Avissar and R Barua, "An Optimal Memory Allocation Scheme for Scratch-Pad-Based Embedded Systems," *ACM Transactions on Embedded Computing Systems,* vol. 1, pp. 6-26, 2002.

[Baco94] D. F. Bacon, S. L. Graham, O. Sharp, ”Compiler transformation for high performance computing,” In *ACM Computing survey*, 26(4), pages345-420, Dec. 1994

[Baga97] A. Baganne, "Méthodologie de synthèse des unités de communication matérielles dans une approche de conception mixte logiciel/matériel (co-design)," Thèse de l'université de Rennes I, décembre 1997

[Baga98] A. Baganne, J. L. Philippe, E. Martin "A Formal Technique for Hardware Interface design," In *IEEE Transactions On Circuits And Systems*, Vol.45, N5, pages 584-591, 1998.

[Bala88] M. Balakrishnan, A. K. Majumdar, D. K. Banerji, J. G. Linders, and J. C. Majithia. ”Allocation of multiport memories in data path synthesis.” In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 7(4):536-540, April 1988.

[Bhat99] A. Bhattacharya et S. K. Das, "LeZi-Update : An Information- Theoretic Approach to Track Mobile Users in PCS Networks," In *Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom'99)*, pages 1-12, 1999

[Broc00] E. Brockmeyer, A. Vandecappelle, F. Catthoor, "Systematic Cycle budget versus System Power Trade-off: a New Perspective on System Exploration of Real-time Data-dominated Applications," In *Proceedings of IEEE Intnl. Symp. on Low Power Design*, pages 137-142, Aug. 2000.

[Cass04] E. Casseau, B. Le Gal, C. Jégo, N. Le Héno, E. Martin, "Reed-Solomon behavioral virtual component for communication systems," In *Proceedings of IEEE International Symposium on Circuits and Systems,* pages 21-23 mai 2004.

[Catt02] F. Catthoor, K. Danckaert, C.Kulkarni, E. Brockmeyer, P.G. Kjeldsberg, T. Van Achteren, T. Omnes,"*Data access and storage management for embedded programmable processors*", ISBN 0-7923-7689-7, Kluwer Acad. Publ., Boston, 2002.

[Catt98] F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, A. Vandecappelle, "*Custom Memory Management Methodology : Exploration of Memory Organisation for Embedded Multimedia System Design*," ISBN 0-7923-8288-9, Kluwer Acad. Publ., Boston,1998.

[Chil97] D. Chillet, "Méthodologie de conception architecturale des mémoires pour circuits dédiés au traitement du signal temps réel," Thèse de Doctorat de l'Université de Rennes I, Janvier 1997

[Chim98] J. H. P. Chim, M. Green, R. W. H. Lau, H. VA Leong et A. SI, "On Caching and Prefetching of Virtual Objects in Distributed Virtual Environments," In *Proceedings of the 6th ACM International Conference on Multimedia (Multimedia'98)*, pages 171-180, sept. 1998

[Cous03a] P. Coussy, A. Baganne, E. Martin, "Communication and Timing Constraints Analysis for IP Design and Integration," In *Proceedings of IFIP WG 10.5 Very Large Scale Integration of System-on-Chip (VLSI-SOC) Conference*, pages 38-43,December 1-3, 2003.

[Cous03b] P. Coussy, "*Synthèse d'Interface de Communication pour les Composants Virtuels*," Thèse de Doctorat de l'Université de Bretagne Sud, Décembre, 2003.

[Danc00] K. Danckaert, F. Catthoor, H. De Man, "A preprocessing step for global loop transformations for data transfer and storage optimization," In *Proceedings of Intnl. Conference on Compilers, Arch. and Synth. for Embedded Systems*, pages 34 - 40, Nov. 2000.

[Digu97] J.P. Diguet, S. Wuytack, F. Catthoor, H. De Man, "Formalized methodology for data reuse exploration in hierarchical memory mappings," In *Proceedings of IEEE Intnl. Symp. on Low Power Design*, pages 30-35, Aug. 1997.

[Flyn72] M. J. Flynn, "Some computer organizations and their effectiveness," In *IEEE Transactions on Computers*, volume C-21, pages 948-960, 1972.

[Gasj91] D. Gajski et al., "High-Level Synthesis : Introduction to Chip and System Design," Kluwer Academic Publishers, 1991

[Grun01] P. Grun, N. Dutt, A. Nicolau," Access Pattern based Local Memory Customization for Low Power Embedded Systems," In *Proceedings of DATE conference*, pages 778-784 , mar 2001

[Henn02] J. Hennessy, D. Patterson," *Architecture des ordinateurs : une approche quantitative*," édition Vuibert ISBN 2-7117-8700-I.

[ITRS03] ITRS Roadmap, url : http://public.itrs.net/

[Imec04] http://www.imec.be/design/atomium/

[Jave03] G. Javel, *"Pratique de la gestion industrielle : Organisation, méthodes et outils,"* ISBN 2100053868, Edition Dunod, Collection Technique et ingénierie, 2003.

[Joup90] N. P. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364-373, juin 1990

[Joup99] N. Jouppi et G. Reinman, "An Integrated Cache Timing and Power Model," *Technical Report 99*, DEC Western Research Laboratory, Summer 1999.

[Kamb97] M. B. Kamble, K. Ghose, ”Analytical energy dissipation models for low power caches,” In *Proceedings of int. Symp. On low power electronic and design*, pages 143-148, aug 1997

[Kand01] M. Kandemir and A. Choudhary, "Compiler-Directed Scratch Pad Memory Hierarchy Design and Management," In *Proceedings of DAC,* pages 628-633, 2002.

[Karc94] D. Karchmer and J. Rose, "Definition and solution of the memory packing problem for field-programmable systems.," In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design*, pages 20-26, November 1994.

[Kim93] T. Kim and C. L. Liu, ”Utilization of multiport memories in data path synthesis.” In *Proceedings of Design Automation Conference*, pages 298-302, June 1993.

[Knap95] D. Knapp, T. Ly, D. MacMillen, R. Mille, "Behavioral synthesis methodology for HDL-based specification and validation," In *Proceedings of Design Automation Conference*, pages 286-291, June 1995.

[Kram92] H. Kramer and J. Muller, "Assignment of global memory elements for multi-process vhdl specifications," In *Proceedings of the International Conference on Computer Aided Design*, pages 496-501, Nov 1992.

[Ku91] D. Ku, G. De Micheli, "Constrained Resource Sharing and Conflict Resolution in Hebe," In *Integration: the VLSI Journal*. Vol. 12, No. 2, pages 131-166, 1991

[Kuen97] G. H. Kuenning, "*Seer : Predictive File Hoarding for Disconnected Mobile Operation*," Thèse de doctorat, University of California, Los Angeles, California, USA, mai 1997

[Kulk01] C. Kulkarni,” Cache optimisation for multimedia application,” thèse feb 2001

[Lee95] H.-D. Lee and S.-Y. Hwang.” A scheduling algorithm for multiport memory minimization in datapath synthesis,” In *Proceedings of the Conference on Asia Pacific Design Automation Conference*, pages 93-100, August 1995.

[Lei97] H. Lei et D. Duchamp, "An Analytical Approach to File Prefetching," In *Proceedings of the USENIX 1997 Annual Technical Conference*, pages 275-288, jan 1997

[Liu95] G. Liu, A. Marlevi et G. Q. maguire Jr, "A Mobile Virtual-Distributed System Architecture for Supporting Wireless Mobile Computing and Communications," In *Proceedings of the 1st Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom'95)*, pages 111-118, nov 1995

[Ly95] T. Ly, D. Knapp, R. Miller, and D. MacMillen, "Scheduling using behavioral templates," In *ACM/IEEE Proceedings of Design Automation Conference*, pages 599-604, June 1995.

[Mart04] F. Marteil, N. Julien, E. Senn, E. Martin, " A Complete Methodology for Memory Optimization in DSP Applications," In *Proceedings of EUROMICRO symposium on Digital System Design*, pages 98-103, sept 2004

[Mart93] E. Martin, O. Sentieys, J. L. Philippe, "GAUT, An Architecture Synthesis Tool for Dedicated Signal Processors," In *Proceedings of EURO-DAC 93*, pp.14-19, 1993

[Mart96] E. Martin, J. L. Philippe, "*Ingénierie des systèmes à microprocesseurs*," ISBN 2-225-85230-8, édition Masson 1996.

[Mass99] K.Masselos, F.Catthoor, C.E.Goutis, H.De Man, "System-level power optimizing data-flow transformations for multimedia applications realized on programmable multimedia processors," In *Proceedings of Intnl. Conference on Electronic Circuits and Systems*, pages 1733-1736, Sep. 1999.

[Ment04] www.mentor.com/c-design/

[Ment98] Mentor Inc. Monet User's and reference manual

[Neer93] M.Neeracher, R.Rühl, "Automatic parallelization of LINPACK routines on distributed memory parallel processors," In *Proceedings of IEEE Int. Parallel Proceedings Symposium*, pages April 1993.

[Nico93] A. Nicolau and S. Novack.” Trailblazing: A hierarchical approach to percolation scheduling,” In *Proceedings of ICPP conference*, pages 120-124, 1993.

[Ohno78] T. Ohno, "Toyota Production System: Beyond Large-Scale Production," 1978

[Ohno90] Taiichi Ohno, "*L'esprit TOYOTA*," édition Masson, Paris, 1990, ISBN:2-225-81709-X

[Pand97a] P. R. Panda, N. D. Dutt, and A. Nicolau.” Memory data organization for improved cache performance in embedded processor applications.” In *ACM Transactions on Design Automation of Electronic Systems*, vol 2(4) pp384-409, October 1997.

[Pand97b] P. R. Panda, N. D. Dutt,” Behavioral array mapping into multiport memories targeting low power,” In *Proceedings of the 10th International Conference on VLSI Design*, pages 268-272, Jan 1997

[Pand97c] P. Panda, N. D. Dutt A. Nicolau,” Exploiting Off-Chip Memory Access Modes in High-Level Synthesis,” *Tech. Rep. #97-32*, U.C. Irvine, 1997

[Pand98] P. R. Panda,” *Memory optimization and exploration for embedded systems*.” Thèse, Université de Californie, Irvine, 1998.

[Pand99a] P. R. Panda, N. D. Dutt, and A. Nicolau, "*Memory Issues in Embedded Systems-On-Chip: Optimizations and Exploration*," ISBN 0-7923-8362-, Kluwer Academic Publishers, Norwell, MA, 1999.

[Pand99b] P. R. Panda, N. D. Dutt, and A. Nicolau.” Local memory exploration and optimization in embedded systems,” In *IEEE Transactions on Computer Aided Design*, 18(1), pages 3-13, January 1999.

[Pass95] N.Passos, E.Sha, L-F.Chao, "Multi-dimensional interleaving for time-and-memory design optimization," In *Proceedings of IEEE Int. Conference On Computer Design*, pages 440-445, Oct. 1995.

[Patt95] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky et J. Zelenka, "Informed Prefetching and Caching," In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP'95)*, pages 79-95, déc 1995

[Phil95] J. L. Philippe, O. Sentieys, J.P. Diguet, E. Martin, "From Digital Signal Processing Specification to Layout," In *Proceedings on Logic and Architecture Synthesis : state-of-the-art and novel approaches*, pages 307-313, 1995

[Pign01] S. Pignolo, E. Martin, N. Julien, E. Senn, B. Saget, "Optimisation de la consommation d’énergie des applications de Traitements Du Signal et de l’Image embarquées sur DSP," In *Proceedings of journées FTFC*, juin 2001

[Pign02] S. Pignolo, "*Optimisation de la consommation des systèmes temps reel embarqués*," Thèse de Doctorat de l'Université de Bretagne Sud, janvier 2002.

[Senn04] E. Senn, J. Laurent, N. Julien, E. Martin, "SoftExplorer: Estimation, Characterization, and Optimization of the Power and Energy Consumption at the Algorithmic Level," In *Proc of PATMOS Conference*, pages 342-351, Sep 2004

[Sloc97] P.Slock, S.Wuytack, F.Catthoor, G.de Jong, "Fast and extensive system-level memory exploration for ATM applications," In *Proceedings of 10th ACM/IEEE Intnl. Symp. on System-Level Synthesis*, pages 74-81, Sep. 1997.

[Sout91] J.Souty, *"L'ingénierie de production 50 fiches pour mieux produire,"* ISBN : 2225823057, Edition MASSON, Collection Organisation Industrielle, 1991

[Stei02] S. Steinke et.al., "Assigning Program and Data Objects to Scratchpad for Energy Reduction," In *Proceeding of DATE conference,* pages 409-414, 2002.

[Stei02a] S. Steinke et.al., "Reducing Energy Consumption by Dynamic Copying of Instructions onto Onchip Memory," In *Proceeding on ISSS,* pages 213-218, 2002.

[Syno03] Synopsys Inc : http://www.synopsys.com/products/products.html

[Syno97] Synopsys Inc., Mountain View, CA. Behavioral Compiler User Guide, 1997.

[Tait95] C. D. Tait, H. Lei, S. Acharya et H. Chang, "Intelligent File Hoarding for Mobile Computers , *Proceedings of the 1st Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom'95)*, pages 119-125, USA, nov 1995

[Tron02] R. Tronçon, G. Janssens, H. Vandecasteele ," Storage Size Reduction by In-Place Mapping of Arrays ," In *Proceedings of International Workshop on Verification, Model Checking and Abstract*, pages 1-12, 2002

[Uday03] S. Udayakumaran and R. Barua, "Compiler-Decided Dynamic Memory Allocation for Scratch-Pad Based Embedded Systems," In *Proceeding of CASES conference,* pages 276-286, 2003.

[Vand99] A.Vandecappelle, M.Miranda, E.Brockmeyer F.Catthoor, D.Verkest, "Global Multimedia System Design Exploration using Accurate Memory Organization Feedback" In *Proceedings 36th ACM/IEEE Design Automation Conference ,* pages 327-332, June1999.

[Verh95] W.Verhaegh, P.Lippens, E.Aarts, J.Korst, J.van Meerbergen, A.van der Werf, "Improved Force-Directed Scheduling in High-Throughput Digital Signal Processing," In *IEEE Transactions on Computer-Aided Design Of Integrated Circuits and Systems*, Volume 14, Number 8, pages 945-960, August 1995

[Verm04] M. Verma, L. Wehmeyer, P. Marwedel, " Dynamic overlay of Srcatchpad Memory for Energy Minimization," In *Proceeding on CODES+ISSS*, pages 104-109, Sep 2004

[Wuyt99] S.Wuytack, F.Catthoor, G.De Jong, and H.De Man. "Minimizing the required memory bandwidth in vlsi system realizations" In *IEEE Transactions on VLSI Systems*, 7(4):pages 433-441, December 1999.

Annexes

Annexe A : Définition de l'ordonnancement dans l'outil GAUT

Nous décrivons les traitements réalisés par la phase d'ordonnancement. Une première étape initialise des structures parmi lesquelles se trouvent deux ensembles de files dont les fonctions sont primordiales. Le premier ensemble permet la gestion des opérations et de leurs différents états : *"exécutables"* et *"en cours" d'exécution*. Une file est attribuée à chaque état et pour chaque type de fonction (addition, soustraction, multiplication...). Lorsque la phase d'ordonnancement débute, l'ensemble des opérations *exécutables* est initialisé avec l'ensemble des opérations dont la date au plus tôt est égale à 0. Lorsqu'une opération de fonction *f* devient *exécutable*, c'est à dire lorsque l'ensemble de ces prédécesseurs, de type opération, a été réalisé, elle est enfilée dans la liste de l'ensemble des opérations de type *f exécutables*. La gestion des opérations *exécutables* est réalisée à l'aide de files ordonnées par la priorité des opérations. Au moment où cette opération est assignée à un opérateur, elle est défilée de l'ensemble des opérations *exécutables*, et enfilée dans l'ensemble des opérations de type *f en cours d'exécution*. Le deuxième ensemble permet la gestion des opérateurs et de leurs différents états: *"sommeil", "en attente", "disponible"* et les sous états "*libre"* et *"occupé"* de l'état *disponible.* Un opérateur *disponible* est *libre* à un instant *t* s'il peut exécuter une opération à cet instant. Il sera *occupé* pendant toute la durée de l'exécution. Les opérateurs en *sommeil* sont des opérateurs qui ont été créés par l'allocation mais qui n'ont encore jamais été utilisés. Les opérateurs qui ont déjà été utilisés (*libres*) sont prioritaires par rapport aux autres opérateurs (en *sommeil*). Les opérateurs sont créés dans l'état *sommeil* à chaque nouvelle tranche d'après le taux de parallélisme calculé lors de l'allocation : l'ensemble des opérateurs *disponibles* est donc vide. Lorsqu'un opérateur est utilisé pour la première fois il passe de l'état *sommeil* à l'état *disponible et occupé*. Lorsqu'un opérateur est *"usé"*, c'est à dire lorsque son temps d'utilisation total devient supérieur au temps de cadence, il est retiré de l'ensemble des opérateurs *disponibles.* Lorsqu'il n'existe plus d'opérateurs *disponibles,* on va chercher l'opérateur souhaité dans l'ensemble des opérateurs en *sommeil*. Les opérateurs *disponibles* sont triés par état et par date de première utilisation, donc par priorité décroissante. Un opérateur est *en attente* à l'instant *t* s'il ne peut terminer l'opération avant le prochain *Tcycle.* Le *Tcycle* est utilisé comme point de (re)synchronisation des opérations dans l'unité de traitement. Ceci limite le nombre de combinaisons d'opérations dans un *Tcycle* et aboutit à la réduction du nombre d'états de la FSM. Les quatre étapes suivantes réalisent le cœur d'ordonnancement et d'assignation de *GAUT :*

1. Sélection d'une opération : les opérations exécutables sont triées suivant leur marge croissante. La marge d'une opération correspond au temps qu'il reste entre l'instant courant et sa date au plus tard.

$$Marge(s) = DTard(s) - t$$

Ainsi, plus la marge est grande, moins l'opération est prioritaire.

2. Sélection d'un opérateur : on tente d'assigner chaque opération exécutable à un opérateur suivant la priorité des opérateurs.

- Opérateur déjà mis en œuvre.
- Opérateur minimisant le nombre de liaisons avec les opérateurs réalisant les opérations précédant l'opération à allouer. Cette gestion des opérateurs permet une optimisation des liaisons inter cellules et réduit donc la complexité des interconnexions.

Lorsqu'une opération n'a pu être assignée, elle est conservée dans l'ensemble des opérations exécutables. Elle sera considérée lors de la prochaine phase d'ordonnancement et ce, pour la date définie par la fonction de progression du temps. Une opération à marge négative est dite *immédiate* et est assignée à un opérateur en *sommeil* s'il n'existe plus d'opérateur *libre*.

3. Progression du temps : la progression du temps est particulière. Dans un premier temps on scrute l'ensemble des opérations en cours d'exécution pour sélectionner le plus petit temps *t1* des dates de relâchement. Cette date de relâchement est égale à la date de fin pour les opérations assignées à des opérateurs non pipeline. Par contre, pour une opération assignée à un opérateur pipeline, la date de relâchement est différente de la date de fin, et correspond à l'instant *t2* où (bien que l'opération ne soit par terminée) le premier étage de pipeline de l'opérateur doit être relâché. Le nouveau temps t est alors égal à $t = \min(t1,t2)$. Lorsque le temps courant *t* dépasse l'instant de la tranche de pipeline suivante *(Tranche*Tcadence)*, on crée alors les opérateurs prévus par l'allocation pour cette tranche. Le temps courant est égal au *Tcycle* suivant lorsqu'il ne reste que des opérations exécutables sur des opérateurs en *attente*.

4. Libération des opérateurs : après avoir fait progresser le temps, on recherche parmi les opérations en cours, celles qui se terminent ou relâchent un opérateur pipeline. Dans ce cas, les opérateurs *occupés* deviennent des opérateurs dits *disponibles*. Lorsqu'une opération *s* est réellement terminée, c'est à dire lorsque sa date de fin est égale au temps courant, on cherche les nouvelles opérations exécutables engendrées par la terminaison de *s*. Pour cela, on recherche parmi tous les successeurs de *s*, les opérations pour lesquelles l'ensemble des prédécesseurs a été réalisé. Les opérateurs *usés* sont retirés de l'ensemble des opérateurs *disponibles*.

Description VHDL des unités de traitement et de mémorisation au niveau RTL d'un filtre LMS 9 points

5. Le fonctionnement pipeline : la synthèse de l'UT peut imposer un mode d'exécution pipeline. L'allocation en moyenne du nombre d'opérateurs à mettre en œuvre, l'utilisation en priorité des opérateurs déjà utilisés ainsi que le retardement de certaines opérations, lié au *Tcycle,* font que le nombre de tranches de pipeline n'est connu qu'après l'ordonnancement.

Annexe B : Fichiers générés par l'outil GAUT

File : lms8.vhd

```
-- FICHIER : lms8.vhd

--  Fichier genere par GAUT 0_4d build 20/10/2004
--  avec la ligne de commande suivante : gaut -gc
lms8 -l C:\GAUT04d\GautC\lib\notech.lib -ng -kt 1
-tl 31250 -cout 0 -optim 0 -const phase_min 10 -
mem -gantt -pipe_off -ob -opt_decoder -mapping
C:\GAUT04d\test\lms8\lms8_2b.map -descr -nvhdl -
java
--
--  Interface entre GAUT et les outils de synthese
logiques
--  Fichier source : lms8
--  Date: Wed Jul 06 15:39:49 2005
--  Resultats de la synthese (0 h 0 mn 0 s):
--  Periode de l'horloge systeme : 10 ns
--  Cadence de l'architecture : 31250 ns   (k,
Tcycle) = (1562, 20 ns)
--  Temps de traversee de l'architecture : 390 ns
--  Nombre de tranches de l'architecture : 1
--  Surface parametree de l'architecture : 1129
--  Surface reelle de l'architecture  : 2975
--    sous16b_vhdl        : 1     (Surface : 52)
--    mult16b_vhdl        : 1     (Surface : 1024)
--   add16b_vhdl  : 1     (Surface : 53)
--    Registres   : 20    (Surface : 70)
--   Multiplexeurs        : 1     (Surface : 24)
--    Demultiplexeurs     : 0     (Surface : 34)
--   TriState     : 17    (Surface : 16)
--    Bus: 3     (Surface : 50)

library IEEE;
use IEEE.STD_LOGIC_1164.all;


ENTITY lms8_UT IS
   PORT (
        BUS_DONNEES_1_lms8 : INOUT
std_logic_vector(15 DOWNTO 0) ;
        BUS_DONNEES_2_lms8 : INOUT
std_logic_vector(15 DOWNTO 0) ;
        BUS_DONNEES_3_lms8 : INOUT
std_logic_vector(15 DOWNTO 0) ;
        CMD_reg16b_vhdl_20 : IN std_logic ;

        .
        .
        .

        CMD_reg16b_vhdl_1 : IN std_logic ;
        CMD_mux16b_vhdl_12 : IN std_logic ;
        CS_tri16b_vhdl_20_1 : IN std_logic ;

        .
        .
        .

        CS_tri16b_vhdl_3_1 : IN std_logic ;
        Rstb : IN std_logic ;
        Clk : IN std_logic ) ;
END ;

ARCHITECTURE lms8_UT_arch OF lms8_UT IS

   COMPONENT reg16b_vhdl
        PORT (
   d : IN std_logic_vector(15 DOWNTO 0) ;
   load : IN std_logic ;
   clrb : IN std_logic ;
   clk : IN std_logic ;
   q : OUT std_logic_vector(15 DOWNTO 0) ) ;
   END COMPONENT ;

   COMPONENT mux16b_vhdl
        PORT (
   e0 : IN std_logic_vector(15 DOWNTO 0) ;
   e1 : IN std_logic_vector(15 DOWNTO 0) ;
   commande : IN std_logic ;
   s : OUT std_logic_vector(15 DOWNTO 0) ) ;
   END COMPONENT ;

   COMPONENT demux16b_vhdl
        PORT (
   e : IN std_logic_vector(15 DOWNTO 0) ;
   commande : IN std_logic ;
   s0 : OUT std_logic_vector(15 DOWNTO 0) ;
   s1 : OUT std_logic_vector(15 DOWNTO 0) ) ;
   END COMPONENT ;

   COMPONENT tri16b_vhdl
        PORT (
   d : IN std_logic_vector(15 DOWNTO 0) ;
   oe : IN std_logic ;
   q : OUT std_logic_vector(15 DOWNTO 0) ) ;
   END COMPONENT ;

   COMPONENT add16b_vhdl
        PORT (
   a : IN std_logic_vector(15 DOWNTO 0) ;
   b : IN std_logic_vector(15 DOWNTO 0) ;
   s : OUT std_logic_vector(15 DOWNTO 0) ) ;
   END COMPONENT ;

   COMPONENT sous16b_vhdl
        PORT (
   a : IN std_logic_vector(15 DOWNTO 0) ;
   b : IN std_logic_vector(15 DOWNTO 0) ;
   s : OUT std_logic_vector(15 DOWNTO 0) ) ;
   END COMPONENT ;

   COMPONENT mult16b_vhdl
        PORT (
   a : IN std_logic_vector(15 DOWNTO 0) ;
   b : IN std_logic_vector(15 DOWNTO 0) ;
   outhigh : OUT std_logic_vector(15 DOWNTO 0) )
;
   END COMPONENT ;

   SIGNAL IN_tri16b_vhdl_3_1 : std_logic_vector
(15 DOWNTO 0) ;

        .
        .
        .

   SIGNAL IN_tri16b_vhdl_20_1 : std_logic_vector
(15 DOWNTO 0) ;
   SIGNAL OUT_sous16b_vhdl_1 : std_logic_vector
(15 DOWNTO 0) ;
   SIGNAL IN_2_sous16b_vhdl_1 : std_logic_vector
(15 DOWNTO 0) ;
   SIGNAL IN_1_sous16b_vhdl_1 : std_logic_vector
(15 DOWNTO 0) ;
   SIGNAL OUT_mult16b_vhdl_2 : std_logic_vector
(15 DOWNTO 0) ;
   SIGNAL IN_2_mult16b_vhdl_2 : std_logic_vector
(15 DOWNTO 0) ;
   SIGNAL IN_1_mult16b_vhdl_2 : std_logic_vector
(15 DOWNTO 0) ;
   SIGNAL OUT_add16b_vhdl_3 : std_logic_vector
(15 DOWNTO 0) ;
   SIGNAL IN_2_add16b_vhdl_3 : std_logic_vector
(15 DOWNTO 0) ;
   SIGNAL IN_1_add16b_vhdl_3 : std_logic_vector
(15 DOWNTO 0) ;

   BEGIN

   Cell_3_add16b : add16b_vhdl
        PORT MAP (
   a => IN_1_add16b_vhdl_3 ,
   b => IN_2_add16b_vhdl_3 ,
   s => OUT_add16b_vhdl_3 ) ;

   Cell_2_mult16b : mult16b_vhdl
        PORT MAP (
   a => IN_1_mult16b_vhdl_2 ,
   b => IN_2_mult16b_vhdl_2 ,
   outhigh => OUT_mult16b_vhdl_2 ) ;

   Cell_1_sous16b : sous16b_vhdl
        PORT MAP (
   a => IN_1_sous16b_vhdl_1 ,
   b => IN_2_sous16b_vhdl_1 ,
   s => OUT_sous16b_vhdl_1 ) ;

   Cell_2_reg16b_20 : reg16b_vhdl
        PORT MAP (
```

```
   d => OUT_add16b_vhdl_3 ,
   q => IN_tri16b_vhdl_20_1 ,
   load => CMD_reg16b_vhdl_20 ,
   clrb => Rstb ,
   clk => Clk ) ;

        ⋮

   Cell_1_reg16b_1 : reg16b_vhdl
        PORT MAP (
   d => BUS_DONNEES_1_lms8 ,
   q => IN_1_sous16b_vhdl_1 ,
   load => CMD_reg16b_vhdl_1 ,
   clrb => Rstb ,
   clk => Clk ) ;

   Cell_2_mux16b_12 : mux16b_vhdl
        PORT MAP (
   e0 => BUS_DONNEES_2_lms8 ,
   e1 => BUS_DONNEES_1_lms8 ,
   s => IN_reg16b_vhdl_12 ,
   commande => CMD_mux16b_vhdl_12 ) ;

   Cell_2_tri16b_20_1 : tri16b_vhdl
        PORT MAP (
   d => IN_tri16b_vhdl_20_1 ,
   q => IN_2_mult16b_vhdl_2 ,
   oe => CS_tri16b_vhdl_20_1 ) ;

        ⋮

   Cell_2_tri16b_3_1 : tri16b_vhdl
        PORT MAP (
   d => IN_tri16b_vhdl_3_1 ,
   q => IN_1_mult16b_vhdl_2 ,
   oe => CS_tri16b_vhdl_3_1 ) ;

   BUS_DONNEES_1_lms8 <= (others => 'Z') ;
   BUS_DONNEES_2_lms8 <= (others => 'Z') ;
   BUS_DONNEES_3_lms8 <= (others => 'Z') ;

END ;


library IEEE;
use IEEE.STD_LOGIC_1164.all;


ENTITY lms8_decoder IS
   PORT (
        enable : IN std_logic ;
        Instructions : IN std_logic_vector(5
DOWNTO 0) ;
        CMD_reg16b_vhdl_20 : OUT std_logic ;

        ⋮

        CMD_reg16b_vhdl_1 : OUT std_logic ;
        CMD_mux16b_vhdl_12 : OUT std_logic ;
        CS_tri16b_vhdl_20_1 : OUT std_logic ;

        ⋮

        CS_tri16b_vhdl_3_1 : OUT std_logic ) ;
END ;

ARCHITECTURE lms8_decoder_arch OF lms8_decoder IS
   signal sigs : std_logic_vector(0 to 37);
BEGIN
   CMD_mux16b_vhdl_12 <= sigs(0);
   CMD_reg16b_vhdl_1 <= sigs(1);

        ⋮

   CMD_reg16b_vhdl_9 <= sigs(20);
   CS_tri16b_vhdl_10_1 <= sigs(21);

        ⋮

   CS_tri16b_vhdl_9_1 <= sigs(37);

   PROCESS (Instructions)
   BEGIN
   CASE Instructions IS
   WHEN "000000" =>
sigs <= "00000000000000010000000000000000000000";
   WHEN "000001" =>
sigs <= "00000000000000100000000000000000000000";
   WHEN "000010" =>
sigs <= "00000000000000000000000000000000110000";
   WHEN "000011" =>
sigs <= "00000000000000101000000000000000110000";
   WHEN "000100" =>
sigs <= "10001000000000000000000000000000101000";
   WHEN "000101" =>
sigs <= "00000000000000100110000000000000101000";
   WHEN "000110" =>
sigs <= "00000000010000000001000100000000001100";
   WHEN "000111" =>
sigs <= "00000100000000000110000100000000001010";
   WHEN "001000" =>
sigs <= "00000000000000000001100000000000101100";
   WHEN "001001" =>
sigs <= "00000000000000000011000000000000101010";
   WHEN "001010" =>
sigs <= "00000000001000000001000010000000001100";
   WHEN "001011" =>
sigs <= "00000010000000000110000010000000001010";
   WHEN "001100" =>
sigs <= "00000000000100000001000001000000001100";
   WHEN "001101" =>
sigs <= "00000000000000000011000000100000001010";
   WHEN "001110" =>
sigs <= "00000000000001000001000100000000000101";
   WHEN "001111" =>
sigs <= "10101001000000000000000100000000000011";
   WHEN "010000" =>
sigs <= "00000000100000000000000100001000000000";
   WHEN "010001" =>
sigs <= "00000000000000000001000010000100000000";
   WHEN "010010" =>
sigs <= "00010000000000000000010000100000001000";
   WHEN "010011" =>
sigs <= "00001000000000000011000000010000001000";
   WHEN "010100" =>
sigs <= "00000000000000000001100000000100100100";
   WHEN "010101" =>
sigs <= "00000000000000000001000000000100100010";
   WHEN "010110" =>
sigs <= "00010000000000000000001000010000001000";
   WHEN "010111" =>
sigs <= "00000000000000000011000000010000001000";
   WHEN "011000" =>
sigs <= "00000000010000000001000010000010000100";
   WHEN "011001" =>
sigs <= "00000000000000000001000001000010000010";
   WHEN "011010" =>
sigs <= "00010000000000000000001001000001000000";
   WHEN "011011" =>
sigs <= "00000000000000000001000000100000001000";
   WHEN "011100" =>
sigs <= "00010000000000000000001100000000001000";
   WHEN "011101" =>
sigs <= "00000000000000000011000010000000001000";
   WHEN "011110" =>
sigs <= "00000000001000000001000000100000000101";
   WHEN "011111" =>
sigs <= "00000000000000000001000000010000000011";
   WHEN "100000" =>
sigs <= "00010000000000000000001000011000000000";
   WHEN "100001" =>
sigs <= "00000000000000000001000000011000000000";
   WHEN "100010" =>
sigs <= "00010000000000000000001100000100000000";
   WHEN "100011" =>
sigs <= "00000000000000000001000010000100000000";
   WHEN "100100" =>
sigs <= "00000000000000000000100100000000000000";
   WHEN "100101" =>
sigs <= "00000000000000000000000000000000000010";
   WHEN "100110" =>
sigs <= "01000000000010000000000000000000000000";
   WHEN OTHERS =>
```

```
sigs <= "01000000000010000000000000000000000000";
   END CASE ;

   END PROCESS ;
END ;
-- Nombre d'etat avec compteur

library IEEE;
use IEEE.STD_LOGIC_1164.all;

ENTITY lms8_statemachine IS
   PORT (       enable : in std_logic;
        Instructions : OUT Std_Logic_Vector(6 - 1
DOWNTO 0);
        Rstb : IN Std_Logic;
        Clk : IN Std_Logic );
END;

ARCHITECTURE lms8_statemachine_arch of
lms8_statemachine IS
   TYPE StateType IS (
        E0, E1, E2, E3, E4, E5, E6, E7, E8, E9,
E10, E11, E12, E13, E14, E15,
        E16, E17, E18, E19, E20, E21, E22, E23,
E24, E25, E26, E27, E28, E29, E30, E31,
        E32, E33, E34, E35, E36, E37, E38 );
BEGIN
   PROCESS(Clk, Rstb)
   VARIABLE CurrentState : StateType;
   VARIABLE NextState : StateType;
   VARIABLE Wait_state : std_logic;
   BEGIN
   IF Rstb = '0' THEN
        Wait_state := '0';
        NextState := E38;
        CurrentState := E38;
        Instructions <= (others => 'X');
   ELSIF Clk'event AND Clk = '1' THEN
        if enable = '1' then CurrentState :=
NextState; end if;
        CASE CurrentState IS
   WHEN E0 =>
-- Temps 0
        Instructions <= "000000" ;
        NextState := E1 ;
   WHEN E1 =>

        ⋮

        ⋮
-- Temps 380
        Instructions <= "100110" ;
        NextState := E0 ;
   WHEN OTHERS =>
        Instructions <= "XXXXXX" ;
        NextState := E0 ;
        END CASE;
   END IF;
   END PROCESS;
END;


library IEEE;
use IEEE.STD_LOGIC_1164.all;


ENTITY lms8_UC IS
   PORT (
        enable : IN std_logic ;
        CMD_reg16b_vhdl_20 : OUT std_logic ;

        ⋮

        CMD_reg16b_vhdl_1 : OUT std_logic ;
        CMD_mux16b_vhdl_12 : OUT std_logic ;
        CS_tri16b_vhdl_20_1 : OUT std_logic ;

        ⋮

        CS_tri16b_vhdl_3_1 : OUT std_logic ;
        Rstb : IN std_logic ;
        Clk : IN std_logic ) ;
END ;

ARCHITECTURE lms8_UC_arch OF lms8_UC IS

   COMPONENT lms8_decoder
        PORT (
   enable : IN std_logic ;
   Instructions : IN std_logic_vector(5 DOWNTO 0)
;
   CMD_reg16b_vhdl_20 : OUT std_logic ;

        ⋮

   CMD_reg16b_vhdl_1 : OUT std_logic ;
   CMD_mux16b_vhdl_12 : OUT std_logic ;
   CS_tri16b_vhdl_20_1 : OUT std_logic ;

        ⋮

   CS_tri16b_vhdl_3_1 : OUT std_logic ) ;
   END COMPONENT ;

   COMPONENT lms8_statemachine
        PORT (   enable : in std_logic;
   Instructions : OUT Std_Logic_Vector(6 - 1
DOWNTO 0);
                Rstb : IN Std_Logic;
   Clk : IN Std_Logic );
   END COMPONENT ;
   SIGNAL Instructions : Std_Logic_Vector(6 - 1
DOWNTO 0) ;

   BEGIN

   controleur : lms8_statemachine
        PORT MAP (
   enable => enable ,
   Rstb => Rstb ,
   Clk => Clk ,
   Instructions => Instructions
    ) ;
   decodeur : lms8_decoder
        PORT MAP (
   enable => enable ,
   Instructions => Instructions ,
   CMD_reg16b_vhdl_20 => CMD_reg16b_vhdl_20 ,

        ⋮

   CMD_reg16b_vhdl_1 => CMD_reg16b_vhdl_1 ,
   CMD_mux16b_vhdl_12 => CMD_mux16b_vhdl_12 ,
   CS_tri16b_vhdl_20_1 => CS_tri16b_vhdl_20_1 ,

        ⋮

   CS_tri16b_vhdl_3_1 => CS_tri16b_vhdl_3_1 ) ;


END ;


library IEEE;
use IEEE.STD_LOGIC_1164.all;


ENTITY lms8 IS
   PORT (
        BUS_DONNEES_1_lms8 : INOUT
std_logic_vector(15 DOWNTO 0) ;
        BUS_DONNEES_2_lms8 : INOUT
std_logic_vector(15 DOWNTO 0) ;
        BUS_DONNEES_3_lms8 : INOUT
std_logic_vector(15 DOWNTO 0) ;
        Rstb : IN std_logic ;
        Clk : IN std_logic ;
        enable : IN std_logic ) ;
END ;
```

```
ARCHITECTURE lms8_arch OF lms8 IS

   COMPONENT lms8_UT
        PORT (
   BUS_DONNEES_1_lms8 : INOUT std_logic_vector(15
DOWNTO 0) ;
   BUS_DONNEES_2_lms8 : INOUT std_logic_vector(15
DOWNTO 0) ;
   BUS_DONNEES_3_lms8 : INOUT std_logic_vector(15
DOWNTO 0) ;
   CMD_reg16b_vhdl_20 : IN std_logic ;

        ⋮

   CMD_reg16b_vhdl_1 : IN std_logic ;
   CMD_mux16b_vhdl_12 : IN std_logic ;
   CS_tri16b_vhdl_20_1 : IN std_logic ;

        ⋮

   CS_tri16b_vhdl_3_1 : IN std_logic ;
   Rstb : IN std_logic ;
   Clk : IN std_logic ) ;
   END COMPONENT ;

   COMPONENT lms8_UC
        PORT (
   CMD_reg16b_vhdl_20 : OUT std_logic ;

        ⋮

   CMD_reg16b_vhdl_1 : OUT std_logic ;
   CMD_mux16b_vhdl_12 : OUT std_logic ;
   CS_tri16b_vhdl_20_1 : OUT std_logic ;

        ⋮

   CS_tri16b_vhdl_3_1 : OUT std_logic ;
   Rstb : IN std_logic ;
   Clk : IN std_logic ;
   enable : IN std_logic ) ;
   END COMPONENT ;

   SIGNAL CS_tri16b_vhdl_3_1_enable : std_logic ;
   SIGNAL CS_tri16b_vhdl_3_1 : std_logic ;

        ⋮

   SIGNAL CS_tri16b_vhdl_20_1_enable :std_logic ;
   SIGNAL CS_tri16b_vhdl_20_1 : std_logic ;
   SIGNAL CMD_mux16b_vhdl_12 : std_logic ;
   SIGNAL CMD_reg16b_vhdl_1_enable : std_logic ;
   SIGNAL CMD_reg16b_vhdl_1 : std_logic ;

        ⋮

   SIGNAL CMD_reg16b_vhdl_20_enable : std_logic ;
   SIGNAL CMD_reg16b_vhdl_20 : std_logic ;

   BEGIN

   UC : lms8_UC
        PORT MAP (
   CMD_reg16b_vhdl_20 => CMD_reg16b_vhdl_20 ,

        ⋮

   CMD_reg16b_vhdl_1 => CMD_reg16b_vhdl_1 ,
   CMD_mux16b_vhdl_12 => CMD_mux16b_vhdl_12 ,
   CS_tri16b_vhdl_20_1 => CS_tri16b_vhdl_20_1 ,

        ⋮

   CS_tri16b_vhdl_3_1 => CS_tri16b_vhdl_3_1 ,
   Rstb => Rstb ,
   Clk => Clk ,
   enable => enable ) ;

   UT : lms8_UT
        PORT MAP (
   BUS_DONNEES_1_lms8 => BUS_DONNEES_1_lms8 ,
   BUS_DONNEES_2_lms8 => BUS_DONNEES_2_lms8 ,
   BUS_DONNEES_3_lms8 => BUS_DONNEES_3_lms8 ,
   CMD_reg16b_vhdl_20 =>
CMD_reg16b_vhdl_20_enable ,
   CMD_reg16b_vhdl_19=>CMD_reg16b_vhdl_19_enable,

        ⋮

   CMD_reg16b_vhdl_1 => CMD_reg16b_vhdl_1_enable,
   CMD_mux16b_vhdl_12 => CMD_mux16b_vhdl_12 ,
   CS_tri16b_vhdl_20_1 =>
CS_tri16b_vhdl_20_1_enable ,

        ⋮

   CS_tri16b_vhdl_3_1 =>
CS_tri16b_vhdl_3_1_enable ,
   Rstb => Rstb ,
   Clk => Clk ) ;

   CS_tri16b_vhdl_3_1_enable <=
CS_tri16b_vhdl_3_1 and enable;

        ⋮

   CS_tri16b_vhdl_20_1_enable <=
CS_tri16b_vhdl_20_1 and enable;
   CMD_reg16b_vhdl_1_enable <= CMD_reg16b_vhdl_1
and enable;

        ⋮

   CMD_reg16b_vhdl_20_enable <=
CMD_reg16b_vhdl_20 and enable;

END ;
```

File : lms8_mu.vhd

```
-- File lms8_mu.vhd generated with
C:/GAUT04d/GautC/bin/mugen
-- LESTER-UBS, 2003
-- Date: Wed Jul 06 15:40:45 2005
-- Command:  C:/GAUT04d/GautC/bin/mugen -i
C:\GAUT04d\test\lms8\lms8.mem -m
C:\GAUT04d\test\lms8\lms8_2b.map -simu -d

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;
library work;
use work.notech.all;
use work.lms8_mu_pack.all;

-- Distributed DPRAM for Xilinx
entity lms8_mu_ram0 is
	generic(width : integer; points : integer; values : BANK0);
	port(
		enable : in  std_logic;
		clk : in  std_logic;
		we  : in  std_logic;
		ra  : in  unsigned(width-1 downto 0);
		wa  : in  unsigned(width-1 downto 0);
		di  : in  word;
		do  : out word
	);
end;

architecture lms8_mu_ram0_arch of lms8_mu_ram0 is
	signal RAM : BANK0 := values;
begin
	process(clk)
	begin
		if clk = '1' and clk'event and enable = '1' then --
sync write
	if we = '1' then
		RAM(to_integer(wa)) <= di;
	end if;
		end if;
	end process;
	do <= RAM(to_integer(ra)); -- async read
end;

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;
library work;
use work.notech.all;
use work.lms8_mu_pack.all;

-- Distributed DPRAM for Xilinx
entity lms8_mu_ram1 is
	generic(width : integer; points : integer; values : BANK1);
	port(
		enable : in  std_logic;
		clk : in  std_logic;
		we  : in  std_logic;
		ra  : in  unsigned(width-1 downto 0);
		wa  : in  unsigned(width-1 downto 0);
		di  : in  word;
		do  : out word
	);
end;

architecture lms8_mu_ram1_arch of lms8_mu_ram1 is
	signal RAM : BANK1 := values;
begin
	process(clk)
	begin
		if clk = '1' and clk'event and enable = '1' then --
sync write
	if we = '1' then
		RAM(to_integer(wa)) <= di;
	end if;
		end if;
	end process;
	do <= RAM(to_integer(ra)); -- async read
end;

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;
library work;
use work.notech.all;
use work.lms8_mu_pack.all;

entity lms8_mu_mux is
	port(
		sel : in std_logic; -- mux command
		e0  : in word; -- first input
		e1  : in word; -- second input
		q : out word -- selected output
	);
end lms8_mu_mux;

architecture lms8_mu_mux_arch of lms8_mu_mux is
	begin
	process(sel, e0, e1)
	begin
		if (sel = '0') then
	q <= e0;
		else
	q <= e1;
		end if;
	end process;
end lms8_mu_mux_arch;

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;
library work;
use work.notech.all;
use work.lms8_mu_pack.all;

entity lms8_mu_megamuxN is
	port(
		di : in warrayN; -- data comes from N buses
		ctrl : in ctrlN; -- to select di(ctrl)
		q : out word -- di(ctrl) selected
	);
end lms8_mu_megamuxN;

architecture lms8_mu_megamuxN_arch of lms8_mu_megamuxN is
	begin
	process(di, ctrl)
	begin
		for i in UM_N-1 downto 0 loop
	if (to_integer(ctrl) = i) then
		q <= di(i); -- only input i selected and output
	end if;
		end loop;
	end process;
end lms8_mu_megamuxN_arch;

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;
library work;
use work.notech.all;
use work.lms8_mu_pack.all;

entity lms8_mu_megamuxP is
	port(
		di : in warrayP; -- data comes from P banks
		ctrl : in ctrlP; -- to select di(ctrl)
		q : out word -- di(ctrl) selected
	);
end lms8_mu_megamuxP;

architecture lms8_mu_megamuxP_arch of lms8_mu_megamuxP is
	begin
	process(di, ctrl)
	begin
		for i in UM_P-1 downto 0 loop
	if (to_integer(ctrl) = i) then
		q <= di(i); -- only input i selected and output
	end if;
		end loop;
	end process;
end lms8_mu_megamuxP_arch;

-- Address generator lms8_mu_gen_access
library IEEE;
```

```
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;
library work;
use work.notech.all;
use work.lms8_mu_pack.all;

entity lms8_mu_gen_access is
	port(
		enable : in std_logic;
		clk  : in std_logic; -- system clock
		rst  : in std_logic; -- system reset, active low
		ra : out warrayRa; -- address of variable in RAM
		wa : out warrayWa; -- address of variable in
RAM
		we : out warrayWe; -- write enable
		sel  : out warraySel;  -- selection for MUX
		cst  : out warrayCst;  -- constant values
		ctrl : out warrayCtrl  -- input bus selector
	);
end lms8_mu_gen_access;

architecture lms8_mu_gen_access_arch of lms8_mu_gen_access is
	type StateType is (S0,…….S38);
	subtype ram_addr_xt is unsigned(4 downto 0);

	type rom_addr_xt is array(0 to 7) of ram_addr_xt;
	type rom_bank_xt is array(0 to 7) of integer;

begin
	process(clk, rst)
	variable curr, nxt : StateType;
	variable oldbase_xt, base_xt : natural; -- aging
	variable addr_ra_xt, addr_wa_xt : natural; -- aging
	variable ADDR_ROM_xt : rom_addr_xt := (
		 to_unsigned(18,5) --@ was 0/18
		,to_unsigned(2,5) --@ was 0/2
		,to_unsigned(3,5) --@ was 0/3
		,to_unsigned(4,5) --@ was 0/4
		,to_unsigned(5,5) --@ was 0/5
		,to_unsigned(6,5) --@ was 0/6
		,to_unsigned(7,5) --@ was 0/7
		,to_unsigned(8,5) --@ was 0/8
	); -- aging
	variable BANK_ROM_xt : rom_bank_xt := ( 0 ,0 ,0 ,0 ,0 ,0
,0 ,0  ); -- aging
	begin
		if rst = '0' then
	curr := S0;
	nxt  := S1;
	ra <= (others => to_unsigned(0,5)); -- @ = 0
	wa <= (others => to_unsigned(0,5)); -- @ = 0
	we <= (others => '0'); -- no write in bank
	sel  <= (others => '0'); -- select a constant
	cst  <= (others =>(others => '0')); -- null constant
	ctrl <= (others =>(others => '0')); -- null control
	base_xt := 0; oldbase_xt := 0;
	addr_ra_xt := 0; addr_wa_xt := 0; -- aging
		-- Following values override the previous ones to
set signals in state S0 at reset (if any)
		ra(0) <= to_unsigned(17,5); -- reading y
		sel(0)  <= '1';
		we(0)  <= '0';
		elsif clk'event and clk = '1' then
	if enable = '1' then
		curr := nxt; -- goto next state
		case curr is
	when S0 => -- time = -10
		ra(0) <= to_unsigned(17,5); -- reading y
		sel(0)  <= '1';
		we(0)  <= '0';                                  nx := S1;
	when S1 => -- time = 0                  nx := S2;
	when S2 => -- time = 10
		sel(0)  <= '0';
		cst(0)  <= int_to_word(0); -- reading 0
		nx := S3;
	when S3 => -- time = 20                 nx := S4;
	when S4 => -- time = 30
		sel(0)  <= '0';
		cst(0)  <= int_to_word(0); -- reading 0
		nx := S5;
	when S5 => -- time = 40
		-- reading x(1) -- aging
		addr_ra_xt := addr(8, base_xt + 1);
		ra( BANK_ROM_xt(addr_ra_xt)) <=
ADDR_ROM_xt(addr_ra_xt);
		sel(BANK_ROM_xt(addr_ra_xt)) <= '1';
		nx := S6;
	when S6 => -- time = 50
		-- reading x(2) -- aging
		addr_ra_xt := addr(8, base_xt + 2);
		ra( BANK_ROM_xt(addr_ra_xt)) <=
ADDR_ROM_xt(addr_ra_xt);
		sel(BANK_ROM_xt(addr_ra_xt)) <= '1';
		ra(1) <= to_unsigned(0,5); -- reading hP0P
		sel(1)  <= '1';                                 nx := S7;
	when S7 => -- time = 60                 nx := S8;
	when S8 => -- time = 70
		-- reading x(3) -- aging
		addr_ra_xt := addr(8, base_xt + 3);
		ra( BANK_ROM_xt(addr_ra_xt)) <=
ADDR_ROM_xt(addr_ra_xt);
		sel(BANK_ROM_xt(addr_ra_xt)) <= '1';
		ra(1) <= to_unsigned(1,5); -- reading hP1P
		sel(1)  <= '1';
		wa(1) <= to_unsigned(0,5); -- reading hP0P
		we(1) <= '1';
		ctrl(1) <= to_unsigned(1,2);
	nx := S9;
	when S9 => -- time = 80
		we(1)  <= '0';                                  nx := S10;
	when S10 => -- time = 90
		ra(1) <= to_unsigned(2,5); -- reading hP2P
		sel(1)  <= '1';
		wa(1) <= to_unsigned(1,5); -- reading hP1P
		we(1) <= '1';
		ctrl(1) <= to_unsigned(1,2);
	nx := S11;
	when S11 => -- time = 100
		we(1)  <= '0';                                  nx := S12;
	when S12 => -- time = 110
		-- reading x(4) -- aging
		addr_ra_xt := addr(8, base_xt + 4);
		ra( BANK_ROM_xt(addr_ra_xt)) <=
ADDR_ROM_xt(addr_ra_xt);
		sel(BANK_ROM_xt(addr_ra_xt)) <= '1';
		ra(1) <= to_unsigned(3,5); -- reading hP3P
		sel(1)  <= '1';
		wa(1) <= to_unsigned(2,5); -- reading hP2P
		we(1) <= '1';
		ctrl(1) <= to_unsigned(1,2);
	nx := S13;
	when S13 => -- time = 120
		we(1)  <= '0';                                  nx := S14;
	when S14 => -- time = 130
		ra(1) <= to_unsigned(4,5); -- reading hP4P
		sel(1)  <= '1';
		wa(1) <= to_unsigned(3,5); -- reading hP3P
		we(1) <= '1';
		ctrl(1) <= to_unsigned(1,2);
	nx := S15;
	when S15 => -- time = 140
		we(1)  <= '0';                                  nx := S16;
	when S16 => -- time = 150
		-- reading x(5) -- aging
		addr_ra_xt := addr(8, base_xt + 5);
		ra( BANK_ROM_xt(addr_ra_xt)) <=
ADDR_ROM_xt(addr_ra_xt);
		sel(BANK_ROM_xt(addr_ra_xt)) <= '1';
		-- reading ___xt_aging_bank_0_addr_18___ --
aging
		addr_wa_xt := addr(8, base_xt + 0);
		wa(  BANK_ROM_xt(addr_wa_xt)) <=
ADDR_ROM_xt(addr_wa_xt);
		we(  BANK_ROM_xt(addr_wa_xt)) <= '1';
		ctrl(BANK_ROM_xt(addr_wa_xt)) <=
to_unsigned(0,2);
		wa(1) <= to_unsigned(4,5); -- reading hP4P
		we(1) <= '1';
		ctrl(1) <= to_unsigned(1,2);
	nx := S17;
	when S17 => -- time = 160
		-- reading x(6) -- aging
		addr_ra_xt := addr(8, base_xt + 6);
		ra( BANK_ROM_xt(addr_ra_xt)) <=
ADDR_ROM_xt(addr_ra_xt);
		sel(BANK_ROM_xt(addr_ra_xt)) <= '1';
		we(BANK_ROM_xt(addr_wa_xt)) <= '0';
		we(1)  <= '0';                                  nx := S18;
	when S18 => -- time = 170                       nx := S19;
	when S19 => -- time = 180                       nx := S20;
	when S20 => -- time = 190
```

```
                -- reading x(7) -- aging
                addr_ra_xt := addr(8, base_xt + 7);
                ra( BANK_ROM_xt(addr_ra_xt)) <=
ADDR_ROM_xt(addr_ra_xt);
                sel(BANK_ROM_xt(addr_ra_xt)) <= '1';
                ra(1) <= to_unsigned(5,5); -- reading hP5P
                sel(1)  <= '1';                         nx := S21;
        when S21 => -- time = 200                       nx := S22;
        when S22 => -- time = 210
                wa(1) <= to_unsigned(5,5); -- reading hP5P
                we(1) <= '1';
                ctrl(1) <= to_unsigned(1,2);
        nx := S23;
        when S23 => -- time = 220
                we(1)  <= '0';                          nx := S24;
        when S24 => -- time = 230
                ra(1) <= to_unsigned(6,5); -- reading hP6P
                sel(1)  <= '1';                         nx := S25;
        when S25 => -- time = 240                       nx := S26;
        when S26 => -- time = 250
                wa(1) <= to_unsigned(6,5); -- reading hP6P
                we(1) <= '1';
                ctrl(1) <= to_unsigned(1,2);
        nx := S27;
        when S27 => -- time = 260
                we(1)  <= '0';                          nx := S28;
        when S28 => -- time = 270                       nx := S29;
        when S29 => -- time = 280                       nx := S30;
        when S30 => -- time = 290
                ra(1) <= to_unsigned(7,5); -- reading hP7P
                sel(1)  <= '1';                         nx := S31;
        when S31 => -- time = 300                       nx := S32;
        when S32 => -- time = 310
                wa(1) <= to_unsigned(7,5); -- reading hP7P
                we(1) <= '1';
                ctrl(1) <= to_unsigned(1,2);
        nx := S33;
        when S33 => -- time = 320
                we(1)  <= '0';                          nx := S34;
        when S34 => -- time = 330                       nx := S35;
        when S35 => -- time = 340                       nx := S36;
        when S36 => -- time = 350                       nx := S37;
        when S37 => -- time = 360                       nx := S38;
        when S38 => -- time = 370
                oldbase_xt := base_xt;
                if (base_xt = 0) then base_xt := 7; else base_xt :=
base_xt - 1; end if; -- aging
                wa(0) <= to_unsigned(17,5); -- reading y
                we(0) <= '1';
                ctrl(0) <= to_unsigned(1,2);
        nx := S0;
                end case;
        end if; -- enable = '1'
                end if;
        end process;
end lms8_mu_gen_access_arch;

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;
library work;
use work.notech.all;
use work.lms8_mu_pack.all;

entity lms8_mu_rambloc0 is
        port (
                enable : in std_logic;
                clk  : in std_logic;          -- CLOCK
                rst  : in std_logic;          -- RESET, active low
                di : in warrayN;              -- DATA in
                do : out word;                          -- DATA
out
                ra : in unsigned(4 downto 0);
                wa : in unsigned(4 downto 0);
                we : in std_logic;
                sel : in std_logic;
                cst : in word;
                ctrl : in ctrlN
        );
end lms8_mu_rambloc0;

architecture lms8_mu_rambloc0_arch of lms8_mu_rambloc0 is

        component lms8_mu_ram0
                generic (width : integer; points : integer; values :
BANK0);
                port (
        enable : in std_logic;
        clk : in std_logic;
        we  : in std_logic;
        ra  : in unsigned(width-1 downto 0);
        wa  : in unsigned(width-1 downto 0);
        di  : in word;
        do  : out word
        );
        end component;

        component lms8_mu_mux
                port (
        sel : in std_logic;
        e0  : in word;
        e1  : in word;
        q : out word
                );
        end component;

        component lms8_mu_megamuxN
                port (
        di : in warrayN;
        ctrl : in ctrlN;
        q : out word
        );
        end component;

        signal mux2ram, ram2mux : word;

begin
        ram : lms8_mu_ram0
                generic map (width => 5, points => 32, values
=> (int_to_word(0), ……..,int_to_word(0)))
                port map (
        enable => enable,  -- system clock
        clk => clk,  -- system clock
        we  => we, -- write enable coming from address generator
        ra  => ra, -- RAM address  ...
        wa  => wa, -- RAM address  ...
        di  => mux2ram, -- is a signal from MEGAMUX to RAM
        do  => ram2mux  -- is a signal from RAM to MUX
                );

        mux : lms8_mu_mux
                port map (
        sel => sel,
        e0  => cst,
        e1  => ram2mux,
        q => do
                );

        megamux : lms8_mu_megamuxN
                port map(
        ctrl => ctrl, -- crtl comes from address generator
        di => di,  -- data comes from rambloc data input
        q => mux2ram -- selected data goes to RAM
                );

end lms8_mu_rambloc0_arch;

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;
library work;
use work.notech.all;
use work.lms8_mu_pack.all;

entity lms8_mu_rambloc1 is
        port (
                enable : in std_logic;
                clk  : in std_logic;          -- CLOCK
                rst  : in std_logic;          -- RESET, active low
                di : in warrayN;              -- DATA in
                do : out word;                          -- DATA
out
                ra : in unsigned(2 downto 0);
                wa : in unsigned(2 downto 0);
                we : in std_logic;
                sel : in std_logic;
                cst : in word;
                ctrl : in ctrlN
        );
end lms8_mu_rambloc1;

architecture lms8_mu_rambloc1_arch of lms8_mu_rambloc1 is
```

```
	component lms8_mu_ram1
		generic (width : integer; points : integer; values :
BANK1);
		port (
	enable : in std_logic;
	clk : in std_logic;
	we  : in std_logic;
	ra  : in unsigned(width-1 downto 0);
	wa  : in unsigned(width-1 downto 0);
	di  : in word;
	do  : out word
	);
	end component;

	component lms8_mu_mux
		port (
	sel : in std_logic;
	e0  : in word;
	e1  : in word;
	q : out word
		);
	end component;

	component lms8_mu_megamuxN
		port (
	di : in warrayN;
	ctrl : in ctrlN;
	q : out word
	);
	end component;

	signal mux2ram, ram2mux : word;

begin
	ram : lms8_mu_ram1
		generic map (width => 3, points => 8, values =>
(int_to_word(0),int_to_word(0),int_to_word(0),int_to_word(0),int_
to_word(0),int_to_word(0),int_to_word(0),int_to_word(0)))
		port map (
	enable => enable,  -- system clock
	clk => clk,  -- system clock
	we  => we, -- write enable coming from address generator
	ra  => ra, -- RAM address  ...
	wa  => wa, -- RAM address  ...
	di  => mux2ram, -- is a signal from MEGAMUX to RAM
	do  => ram2mux  -- is a signal from RAM to MUX
		);

	mux : lms8_mu_mux
		port map (
	sel => sel,
	e0  => cst,
	e1  => ram2mux,
	q => do
		);

	megamux : lms8_mu_megamuxN
		port map(
	ctrl => ctrl, -- crtl comes from address generator
	di => di,  -- data comes from rambloc data input
	q => mux2ram -- selected data goes to RAM
		);

end lms8_mu_rambloc1_arch;

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;
library work;
use work.notech.all;
use work.lms8_mu_pack.all;

entity lms8_mu_fsm1 is
	port(
		enable : in std_logic;
		clk : in std_logic;  -- system clock
		rst : in std_logic;  -- system reset, active low
		oe  : out std_logic; -- to drive tristate buffer
		ctrl : out ctrlP -- to drive input bank selector
	);
end lms8_mu_fsm1;

architecture lms8_mu_fsm1_arch of lms8_mu_fsm1 is
	type StateType is (S0,……..,S38);
	type rom_bank_xt is array(0 to 7) of integer;
begin
	process(clk, rst)
	variable curr, nxt : StateType;
	variable oldbase_xt, base_xt : natural; -- aging
	variable addr_ra_xt : natural; -- aging
	variable BANK_ROM_xt : rom_bank_xt := ( 0 ,0 ,0 ,0 ,0 ,0
,0 ,0  ); -- aging

		begin
			if rst = '0' then
	nxt := S1;
	curr := S0;
	oe <= '0';
	ctrl  <= (others => '0');
	base_xt := 0; addr_ra_xt := 0;
			-- Following values override the previous ones to
set signals in state S0 at reset (if any)		nx := S1;
			elsif clk'event and clk = '1' then
	if enable = '1' then
			curr := nxt;
			case curr IS
	when S0 => -- -10		nx := S1;
	when S1 => -- 0		nx := S2;
	when S2 => -- 10		nx := S3;
	when S3 => -- 20		nx := S4;
	when S4 => -- 30		nx := S5;
	when S5 => -- 40
			oe <= '1';
			addr_ra_xt := addr(8, base_xt + 1);
			ctrl <=
to_unsigned(BANK_ROM_xt(addr_ra_xt), 1); --select bank
			nx := S6;
	when S6 => -- 50
			oe <= '1';
			ctrl <= to_unsigned(1,1); -- select bank 1
			nx := S7;
	when S7 => -- 60
			oe <= '0';		nx := S8;
	when S8 => -- 70
			oe <= '1';
			ctrl <= to_unsigned(1,1); -- select bank 1
			nx := S9;
	when S9 => -- 80
			oe <= '0';		nx := S10;
	when S10 => -- 90
			oe <= '1';
			ctrl <= to_unsigned(1,1); -- select bank 1
			nx := S11;
	when S11 => -- 100
			oe <= '0';		nx := S12;
	when S12 => -- 110
			oe <= '1';
			ctrl <= to_unsigned(1,1); -- select bank 1
			nx := S13;
	when S13 => -- 120
			oe <= '0';		nx := S14;
	when S14 => -- 130
			oe <= '1';
			ctrl <= to_unsigned(1,1); -- select bank 1
			nx := S15;
	when S15 => -- 140
			oe <= '0';		nx := S16;
	when S16 => -- 150		nx := S17;
	when S17 => -- 160
			oe <= '1';
			addr_ra_xt := addr(8, base_xt + 6);
			ctrl <=
to_unsigned(BANK_ROM_xt(addr_ra_xt), 1); --select bank
			nx := S18;
	when S18 => -- 170
			oe <= '0';		nx := S19;
	when S19 => -- 180		nx := S20;
	when S20 => -- 190
			oe <= '1';
			ctrl <= to_unsigned(1,1); -- select bank 1
			nx := S21;
	when S21 => -- 200
			oe <= '0';		nx := S22;
	when S22 => -- 210		nx := S23;
	when S23 => -- 220		nx := S24;
	when S24 => -- 230
			oe <= '1';
			ctrl <= to_unsigned(1,1); -- select bank 1
			nx := S25;
	when S25 => -- 240
```

```
                oe <= '0';                  nx := S26;
        when S26 => -- 250                  nx := S27;
        when S27 => -- 260                  nx := S28;
        when S28 => -- 270                  nx := S29;
        when S29 => -- 280                  nx := S30;
        when S30 => -- 290
                oe <= '1';
                ctrl <= to_unsigned(1,1); -- select bank 1
                nx := S31;
        when S31 => -- 300
                oe <= '0';                  nx := S32;
        when S32 => -- 310                  nx := S33;
        when S33 => -- 320                  nx := S34;
        when S34 => -- 330                  nx := S35;
        when S35 => -- 340                  nx := S36;
        when S36 => -- 350                  nx := S37;
        when S37 => -- 360                  nx := S38;
        when S38 => -- 370
                if (base_xt = 0) then base_xt := 7; else base_xt :=
base_xt -1; end if;             nx := S0;
                end case;
        end if; -- enable = '1'
                end if;
        end process;
end lms8_mu_fsm1_arch;

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;
library work;
use work.notech.all;
use work.lms8_mu_pack.all;

entity lms8_mu_busbloc1 is
        port (
                enable : in std_logic;
                clk  : in std_logic;        -- CLOCK
                rst  : in std_logic;        -- RESET    -- active
low
                data : inout word;          -- DATA in and out,
BUS side
                di : in warrayP;            -- DATA out, RAM
side
                do : out word                           -- DATA
in, RAM side
        );
end lms8_mu_busbloc1;

architecture lms8_mu_busbloc1_arch of lms8_mu_busbloc1 is

        component lms8_mu_mux
                port (
        sel : in std_logic;
        e0  : in word;
        e1  : in word;
        q : out word
                );
        end component;

        component lms8_mu_megamuxP
                port (
        di : in warrayP;
        ctrl : in ctrlP;
        q : out word
        );
        end component;

        component lms8_mu_fsm1
                port (
        enable : in std_logic;
        clk : in std_logic;
        rst : in std_logic;             -- active low
        oe  : out std_logic;
        ctrl : out ctrlP
        );
        end component;

        signal oe : std_logic;
        signal ctrl : ctrlP;
        signal mux2tri : word;

begin
        tri : tri16b_vhdl
                port map(
        d  => mux2tri,  -- input coming from RAM side
        oe => oe, -- output enable command coming from control
FSM
        q  => data -- output going to bus side
                );

        do <= data;
        megamux : lms8_mu_megamuxP
                port map(
        ctrl => ctrl, -- ctrl comes from control FSM
        di => di,  -- data comes from all banks
        q => mux2tri -- selected output goes to RW-mux
                );

        control_fsm : lms8_mu_fsm1
                port map(
        enable => enable,
        clk => clk,  -- system clock
        rst => rst,  -- system reset, active low
        oe  => oe, -- ...
        ctrl => ctrl  -- ...
                );

end lms8_mu_busbloc1_arch;

        .
        .
        .

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;
library work;
use work.notech.all;
use work.lms8_mu_pack.all;

entity lms8_mu_fsm3 is
        port(
                enable : in std_logic;
                clk : in std_logic;  -- system clock
                rst : in std_logic;  -- system reset, active low
                oe  : out std_logic; -- to drive tristate buffer
                ctrl : out ctrlP -- to drive input bank selector
        );
end lms8_mu_fsm3;

architecture lms8_mu_fsm3_arch of lms8_mu_fsm3 is
        type StateType is (S0,……,S38);
        type rom_bank_xt is array(0 to 7) of integer;

begin
        process(clk, rst)
        variable curr, nxt : StateType;
        variable oldbase_xt, base_xt : natural; -- aging
        variable addr_ra_xt : natural; -- aging
        variable BANK_ROM_xt : rom_bank_xt := ( 0 ,0 ,0 ,0 ,0 ,0
,0 ,0  ); -- aging

        begin
                if rst = '0' then
        nxt := S1;
        curr := S0;
        oe <= '0';
        ctrl  <= (others => '0');
        base_xt := 0; addr_ra_xt := 0;
                -- Following values override the previous ones to
set signals in state S0 at reset (if any)               nx := S1;
                elsif clk'event and clk = '1' then
        if enable = '1' then
                curr := nxt;
                case curr IS
        when S0 => -- -10               nx := S1;
        when S1 => -- 0                 nx := S2;
        when S2 => -- 10                nx := S3;
        when S3 => -- 20                nx := S4;
        when S4 => -- 30                nx := S5;
        when S5 => -- 40                nx := S6;
        when S6 => -- 50                nx := S7;
        when S7 => -- 60                nx := S8;
        when S8 => -- 70
                oe <= '1';
                addr_ra_xt := addr(8, base_xt + 3);
                ctrl <=
to_unsigned(BANK_ROM_xt(addr_ra_xt), 1); --select bank
                nx := S9;
        when S9 => -- 80
                oe <= '0';              nx := S10;
        when S10 => -- 90               nx := S11;
```

```
        when S11 => -- 100                      nx := S12;
        when S12 => -- 110
                        oe <= '1';
                        addr_ra_xt := addr(8, base_xt + 4);
                        ctrl <=
to_unsigned(BANK_ROM_xt(addr_ra_xt), 1); --select bank
                        nx := S13;
        when S13 => -- 120
                        oe <= '0';              nx := S14;
        when S14 => -- 130                      nx := S15;
        when S15 => -- 140                      nx := S16;
        when S16 => -- 150
                        oe <= '1';
                        addr_ra_xt := addr(8, base_xt + 5);
                        ctrl <=
to_unsigned(BANK_ROM_xt(addr_ra_xt), 1); --select bank
                        nx := S17;
        when S17 => -- 160
                        oe <= '0';              nx := S18;
        when S18 => -- 170                      nx := S19;
        when S19 => -- 180                      nx := S20;
        when S20 => -- 190                      nx := S21;
        when S21 => -- 200                      nx := S22;
        when S22 => -- 210                      nx := S23;
        when S23 => -- 220                      nx := S24;
        when S24 => -- 230                      nx := S25;
        when S25 => -- 240                      nx := S26;
        when S26 => -- 250                      nx := S27;
        when S27 => -- 260                      nx := S28;
        when S28 => -- 270                      nx := S29;
        when S29 => -- 280                      nx := S30;
        when S30 => -- 290                      nx := S31;
        when S31 => -- 300                      nx := S32;
        when S32 => -- 310                      nx := S33;
        when S33 => -- 320                      nx := S34;
        when S34 => -- 330                      nx := S35;
        when S35 => -- 340                      nx := S36;
        when S36 => -- 350                      nx := S37;
        when S37 => -- 360                      nx := S38;
        when S38 => -- 370
                        if (base_xt = 0) then base_xt := 7; else base_xt :=
base_xt -1; end if;             nx := S0;
                        end case;
        end if; -- enable = '1'
                        end if;
        end process;
end lms8_mu_fsm3_arch;

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;
library work;
use work.notech.all;
use work.lms8_mu_pack.all;

entity lms8_mu_busbloc3 is
        port (
                        enable : in std_logic;
                        clk  : in std_logic;        -- CLOCK
                        rst  : in std_logic;        -- RESET    -- active
low
                        data : inout word;          -- DATA in and out,
BUS side
                        di : in warrayP;            -- DATA out, RAM
side
                        do : out word                           -- DATA
in, RAM side
        );
end lms8_mu_busbloc3;

architecture lms8_mu_busbloc3_arch of lms8_mu_busbloc3 is

        component lms8_mu_mux
                        port (
        sel : in std_logic;
        e0  : in word;
        e1  : in word;
        q : out word
                        );
        end component;

        component lms8_mu_megamuxP
                        port (
        di : in warrayP;
        ctrl : in ctrlP;
        q : out word
        );
        end component;

        component lms8_mu_fsm3
                        port (
        enable : in std_logic;
        clk : in std_logic;
        rst : in std_logic;             -- active low
        oe  : out std_logic;
        ctrl : out ctrlP
        );
        end component;

        signal oe : std_logic;
        signal ctrl : ctrlP;
        signal mux2tri : word;

begin
        tri : tri16b_vhdl
                        port map(
        d  => mux2tri,  -- input coming from RAM side
        oe => oe, -- output enable command coming from control
FSM
        q  => data -- output going to bus side
                        );

        do <= data;
        megamux : lms8_mu_megamuxP
                        port map(
        ctrl => ctrl, -- ctrl comes from control FSM
        di => di,  -- data comes from all banks
        q => mux2tri -- selected output goes to RW-mux
                        );

        control_fsm : lms8_mu_fsm3
                        port map(
        enable => enable,
        clk => clk,  -- system clock
        rst => rst,  -- system reset, active low
        oe  => oe, -- ...
        ctrl => ctrl  -- ...
                        );

end lms8_mu_busbloc3_arch;

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;
library work;
use work.notech.all;
use work.lms8_mu_pack.all;

entity lms8_mu is
        port (
                        enable : in std_logic;
                        clk : in std_logic;         -- CLOCK
                        rst : in std_logic;         -- RESET, active low
                        data : inout warrayN        -- N buses
        );
end lms8_mu;

architecture lms8_mu_arch of lms8_mu is

        component lms8_mu_gen_access
                        port (
        enable : in std_logic;
        clk  : in std_logic;
        rst  : in std_logic;            -- active low
        ra : out warrayRa;
        wa : out warrayWa;
        we : out warrayWe;
        sel  : out warraySel;
        cst  : out warrayCst;
        ctrl : out warrayCtrl
                        );
        end component;

        component lms8_mu_rambloc0
                        port (
        enable : in std_logic;
        clk : in std_logic;
        rst : in std_logic;             -- active low
        di  : in warrayN;
        do  : out word;
        ra : in unsigned(4 downto 0);
        wa : in unsigned(4 downto 0);
```

```
we : in std_logic;
sel : in std_logic;
cst : in word;
ctrl : in ctrlN
        );
end component;

component lms8_mu_rambloc1
        port (
enable : in std_logic;
clk : in std_logic;
rst : in std_logic;          -- active low
di  : in warrayN;
do  : out word;
ra : in unsigned(2 downto 0);
wa : in unsigned(2 downto 0);
we : in std_logic;
sel : in std_logic;
cst : in word;
ctrl : in ctrlN
        );
end component;

component lms8_mu_busbloc1
        port (
enable : in std_logic;
clk  : in std_logic;
rst  : in std_logic;         -- active low
data : inout word;
di  : in warrayP;
do  : out word
);
end component;

component lms8_mu_busbloc2
        port (
enable : in std_logic;
clk  : in std_logic;
rst  : in std_logic;         -- active low
data : inout word;
di  : in warrayP;
do  : out word
);
end component;

component lms8_mu_busbloc3
        port (
enable : in std_logic;
clk  : in std_logic;
rst  : in std_logic;         -- active low
data : inout word;
di  : in warrayP;
do  : out word
);
end component;

signal out_ram0 : word;  -- output RAM 0
signal out_ram1 : word;  -- output RAM 1
signal out_bus1 : word;  -- output BUS 1
signal out_bus2 : word;  -- output BUS 2
signal out_bus3 : word;  -- output BUS 3

signal ra : warrayRa;
signal wa : warrayWa;
signal we : warrayWe;
signal sel : warraySel;
signal cst : warrayCst;
signal ctrl : warrayCtrl;


begin

lms8_mu_gen_access_comp : lms8_mu_gen_access
        port map (
enable => enable,
clk => clk,
rst => rst,
ra => ra,
wa => wa,
we => we,
sel => sel,
cst => cst,
ctrl => ctrl
        );

lms8_mu_rambloc0_comp : lms8_mu_rambloc0
        port map (
enable => enable,
clk => clk,
rst => rst,
ra => ra(0)(4 downto 0),
wa => wa(0)(4 downto 0),
we => we(0),
sel => sel(0),
cst => cst(0),
ctrl => ctrl(0),
di(0)  => out_bus1,
di(1)  => out_bus2,
di(2)  => out_bus3,
do  => out_ram0
        );

lms8_mu_rambloc1_comp : lms8_mu_rambloc1
        port map (
enable => enable,
clk => clk,
rst => rst,
ra => ra(1)(2 downto 0),
wa => wa(1)(2 downto 0),
we => we(1),
sel => sel(1),
cst => cst(1),
ctrl => ctrl(1),
di(0)  => out_bus1,
di(1)  => out_bus2,
di(2)  => out_bus3,
do  => out_ram1
        );

lms8_mu_busbloc1_comp : lms8_mu_busbloc1
        port map (
enable => enable,
clk  => clk,
rst  => rst,
data => data(0),
di(0)  => out_ram0,
di(1)  => out_ram1,
do  => out_bus1
        );

lms8_mu_busbloc2_comp : lms8_mu_busbloc2
        port map (
enable => enable,
clk  => clk,
rst  => rst,
data => data(1),
di(0)  => out_ram0,
di(1)  => out_ram1,
do  => out_bus2
        );

lms8_mu_busbloc3_comp : lms8_mu_busbloc3
        port map (
enable => enable,
clk  => clk,
rst  => rst,
data => data(2),
di(0)  => out_ram0,
di(1)  => out_ram1,
do  => out_bus3
        );

end lms8_mu_arch;

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;
library work;
use work.notech.all;
use work.lms8_mu_pack.all;

entity lms8_mu_du is
        port (
        BUS_DONNEES_1_lms8 : inout std_logic_vector(15
downto 0);
        BUS_DONNEES_2_lms8 : inout std_logic_vector(15
downto 0);
        BUS_DONNEES_3_lms8 : inout std_logic_vector(15
downto 0);
                enable : in std_logic;
                clk : in std_logic;          -- CLOCK
                Rstb : in std_logic          -- RESET, active low
```

```
	);
end lms8_mu_du;

architecture lms8_mu_du_arch of lms8_mu_du is

begin
	BUS_DONNEES_1_lms8 <= (others => 'Z');
	BUS_DONNEES_2_lms8 <= (others => 'Z');
	BUS_DONNEES_3_lms8 <= (others => 'Z');
	process(clk, rstb)
		type TYPE_STATE is (
	S0,S1,S2,S3,S4,S5,S6,S7,S8,S9,S10,S11,S12,S13,S14,S15
	,S16,S17,S18,S19,S20,S21,S22,S23,S24,S25,S26,S27,S28,S2
9,S30,S31
	,S32,S33,S34,S35,S36,S37,S38
		);
		variable cur, nxt : TYPE_STATE;
	begin
		if rstb = '0' then
	cur := S0;
	nxt := S1;
		elsif clk = '1' and clk'event and enable = '1' then
	cur := nxt;
	case cur is
	when S0 =>		nx := S1;
	when S1 =>		nx := S2;
	when S2 =>		nx := S3;
	when S3 =>		nx := S4;
	when S4 =>		nx := S5;
	when S5 =>		nx := S6;
	when S6 =>		nx := S7;
	when S7 =>		nx := S8;
	when S8 =>		nx := S9;
	when S9 =>		nx := S10;
	when S10 =>			nx := S11;
	when S11 =>			nx := S12;
	when S12 =>			nx := S13;
	when S13 =>			nx := S14;
	when S14 =>			nx := S15;
	when S15 =>			nx := S16;
	when S16 =>			nx := S17;
	when S17 =>			nx := S18;
	when S18 =>			nx := S19;
	when S19 =>			nx := S20;
	when S20 =>			nx := S21;
	when S21 =>			nx := S22;
	when S22 =>			nx := S23;
	when S23 =>			nx := S24;
	when S24 =>			nx := S25;
	when S25 =>			nx := S26;
	when S26 =>			nx := S27;
	when S27 =>			nx := S28;
	when S28 =>			nx := S29;
	when S29 =>			nx := S30;
	when S30 =>			nx := S31;
	when S31 =>			nx := S32;
	when S32 =>			nx := S33;
	when S33 =>			nx := S34;
	when S34 =>			nx := S35;
	when S35 =>			nx := S36;
	when S36 =>			nx := S37;
	when S37 =>			nx := S38;
	when S38 =>			nx := S0;
	end case;
		end if;
	end process;
end lms8_mu_du_arch;

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;
library work;
use work.notech.all;
use work.lms8_mu_pack.all;

entity lms8_mu_top is		-- TOP level IC
	port (
		BUS_DONNEES_1_lms8 : inout
std_logic_vector(15 downto 0);
		BUS_DONNEES_2_lms8 : inout
std_logic_vector(15 downto 0);
		BUS_DONNEES_3_lms8 : inout
std_logic_vector(15 downto 0);
		enable : in std_logic;
		clk : in std_logic;		-- CLOCK
		Rstb : in std_logic		-- RESET, active low
	);
end lms8_mu_top;

architecture lms8_mu_top_arch of lms8_mu_top is

	component lms8 -- UT+UC are generated by GAUT
		port (
	enable : in std_logic;
	BUS_DONNEES_1_lms8 : inout std_logic_vector(15
downto 0);
	BUS_DONNEES_2_lms8 : inout std_logic_vector(15
downto 0);
	BUS_DONNEES_3_lms8 : inout std_logic_vector(15
downto 0);
	clk : in std_logic;		-- CLOCK
	Rstb  : in std_logic		-- RESET, active low
		);
	end component;

	component lms8_mu_du -- UT+UC are generated by GAUT
		port (
	enable : in std_logic;
	BUS_DONNEES_1_lms8 : inout std_logic_vector(15
downto 0);
	BUS_DONNEES_2_lms8 : inout std_logic_vector(15
downto 0);
	BUS_DONNEES_3_lms8 : inout std_logic_vector(15
downto 0);
	clk : in std_logic;		-- CLOCK
	Rstb  : in std_logic		-- RESET, active low
		);
	end component;

	component lms8_mu
		port(
	enable : in std_logic;  -- enable
	clk : in std_logic;		-- CLOCK
	rst : in std_logic;		-- RESET OUT, active low
	data : inout warrayN		-- data IN/OUT
		);
	end component;

	signal uc_enable : std_logic;

begin

	lms8_UT_comp : lms8
		port map (
	BUS_DONNEES_1_lms8 => BUS_DONNEES_1_lms8,
	BUS_DONNEES_2_lms8 => BUS_DONNEES_2_lms8,
	BUS_DONNEES_3_lms8 => BUS_DONNEES_3_lms8,
	enable => uc_enable,
	clk => clk,
	Rstb => Rstb
		);

	lms8_mu_comp : lms8_mu
		port map (
	data(0) => BUS_DONNEES_1_lms8,
	data(1) => BUS_DONNEES_2_lms8,
	data(2) => BUS_DONNEES_3_lms8,
	enable => enable,
	clk => clk,
	rst => Rstb
		);

	lms8_mu_du_comp : lms8_mu_du
		port map (
	BUS_DONNEES_1_lms8 => BUS_DONNEES_1_lms8,
	BUS_DONNEES_2_lms8 => BUS_DONNEES_2_lms8,
	BUS_DONNEES_3_lms8 => BUS_DONNEES_3_lms8,
	enable => uc_enable,
	clk => clk,
	Rstb => Rstb
		);

	uc_enable <= enable;

end lms8_mu_top_arch;

-- end of file lms8_mu.vhd, generated by
C:/GAUT04d/GautC/bin/mugen
```

File :
lms8.mem, rapport sur les accès aux mémoires

```
-- HISTOGRAMME DES ACCES MEMOIRE<->REGISTRE

-- Total des acces  memoire -> registre :      18
-- Total des acces registre -> memoire  :      9
-- Total des acces  U.C. -> registre :   2
-- Total des acces registre -> U.C.  :   1
-- Total des acces U.C.  -> memoire  :   0

 PHASE_MIN  = 10 ;
 ACCESMEMOIRE     = 10 ;
 TpsCalcul  = 31250 ;
 NbTranchePip     = 1 ;
 nb_bus     = 3 ;
 nb_bits    = 16 ;
 TpsTraversee     = 390 ;


-- ADRESSE DES CASES MEMOIRE DE L'UNITE DE MEMORISATION


LISTEPOINTS : 21

 Adresse  Lecture  Ecriture     Type     Evolution  TpsDebut  TpsFin  Banc/adr;

    y         1         1     Adaptation      y          0       380      0/17;
 deux_mu      1         0      Constante   deux_mu       0     31250      0/0;
   x(0)       1         0      Constante     x(0)        0     31250      0/1;
   x(2)       1         0       Vieillis     x(1)        0     31250      0/3;
   h(0)       1         1     Adaptation    h(0)       60       80      1/0;
   h(1)       1         1     Adaptation    h(1)       80      100      1/1;
   h(2)       1         1     Adaptation    h(2)      100      120      1/2;
   h(3)       1         1     Adaptation    h(3)      120      140      1/3;
   h(4)       1         1     Adaptation    h(4)      140      160      1/4;
   h(5)       1         1     Adaptation    h(5)      200      220      1/5;
   h(6)       1         1     Adaptation    h(6)      240      260      1/6;
   h(7)       1         1     Adaptation    h(7)      300      320      1/7;
   x(1)       1         0       Vieillis      xt         0     31250      0/2;
   x(7)       1         0       Vieillis    x(6)         0     31250      0/8;
   x(3)       1         0       Vieillis    x(2)         0     31250      0/4;
   x(4)       1         0       Vieillis    x(3)         0     31250      0/5;
   x(5)       1         0       Vieillis    x(4)         0     31250      0/6;
   x(6)       1         0       Vieillis    x(5)         0     31250      0/7;

      -- Total du nombre de points memoire de l'U.M. : 18

-- ADRESSE DES CASES MEMOIRE DE L'UNITE DE COMMUNICATION

  Adresse    Lecture    Ecriture     Type     Evolution   TpsDebut    TpsFin

    yt          1          0       Port_in       yt          0          0;
   yft          0          1      Port_Out      yft        380        380;
    xt          1          0       Port_in       xt        160        160;

      -- Total du nombre de points memoire de l'U.C. : 3
```

Debut	Fin	NumBus	NumReg	Sens	Adresse	Tranche	ASAP	ALAP
LISTEACCES:30								
-10	0	2	2	Lecture	y	1	-10	0;
370	380	2	8	Ecriture	y	1	370	31250;
10	20	2	3	Lecture	deux_mu	1	-10	20;
30	40	2	3	Lecture	x(0)	1	30	40;
50	60	2	3	Lecture	x(2)	1	50	60;
50	60	1	6	Lecture	h(0)	1	-10	60;
70	80	2	8	Ecriture	h(0)	1	70	90;
70	80	1	6	Lecture	h(1)	1	70	80;
90	100	2	8	Ecriture	h(1)	1	90	110;
90	100	1	6	Lecture	h(2)	1	90	100;
110	120	2	8	Ecriture	h(2)	1	110	130;
110	120	1	6	Lecture	h(3)	1	110	120;
130	140	2	8	Ecriture	h(3)	1	130	150;
130	140	1	6	Lecture	h(4)	1	130	140;
150	160	2	8	Ecriture	h(4)	1	150	210;
190	200	1	6	Lecture	h(5)	1	150	200;
210	220	2	8	Ecriture	h(5)	1	210	250;
230	240	1	6	Lecture	h(6)	1	210	240;
250	260	2	8	Ecriture	h(6)	1	250	310;
290	300	1	6	Lecture	h(7)	1	250	300;
310	320	2	8	Ecriture	h(7)	1	310	370;
40	50	1	12	Lecture	x(1)	1	-10	50;
190	200	2	12	Lecture	x(7)	1	180	200;
70	80	3	13	Lecture	x(3)	1	-10	80;
110	120	3	14	Lecture	x(4)	1	-10	120;
150	160	3	15	Lecture	x(5)	1	-10	160;
160	170	1	16	Lecture	x(6)	1	-10	170;
-10	0	1	1	Lecture	yt	1	-10	0;
370	380	2	8	Ecriture	yft	1	370	31250;
150	160	1	12	Lecture	xt	1	150	160;

File :
lms8.out, rapport sur l'architecture générée

```
*****************************************************
Fichier genere par GAUT 0_4d build 20/10/2004
*****************************************************
Date de la Synthese : Date: Wed Jul 06 15:39:49 2005
Algorithme : lms8.src
Nombre de lignes : 126, a plat : 126
Nombre de noeuds du GFD : 88, d'arcs : 168

Chemin critique du GFD: 150 ns
Boucle critique du GFD: 150 ns
Temps sequentiel multifonction du GFD : 500 ns
Temps sequentiel monofonction du GFD : 340 ns

*****************************************************
Parametres de la synthese :
   critere de cout = Operateurs
   optimisation des registres a posteriori = Aucune

*****************************************************
Selection des operateurs :

   reg16b_vhdl, Surface: 70, Operation(s) : reg  (3 ns)
   mux16b_vhdl, Surface: 24, Operation(s) : mux  (10 ns)
   demux16b_vhdl, Surface: 34, Operation(s) : demux  (2 ns)
   tri16b_vhdl, Surface: 16, Operation(s) : tri  (3 ns)
   add16b_vhdl, Surface: 53, Operation(s) : add  (10 ns)
   sous16b_vhdl, Surface: 52, Operation(s) : sub  (10 ns)
   mult16b_vhdl, Surface: 1024, Operation(s) : mul  (20 ns)

*****************************************************
Allocation pipeline ASAP :

   Tranche Numero 1 -->
   Fonction sub :   (Nb = 1, Tl = 10 ns) 10 -->  0.00
   Fonction mul :   (Nb = 17, Tl = 20 ns) 340 -->  0.01
   Fonction add :   (Nb = 15, Tl = 10 ns) 150 -->  0.00

*****************************************************
Resultats de la synthese (0 h 0 mn 0 s):

Periode de l'horloge systeme         : 10 ns
Cadence de l'architecture            : 31250 ns     (k, Tcycle) = (1562, 20 ns)
Temps de traversee de l'architecture : 390 ns
Nombre de tranches de l'architecture : 1

Surface parametree de l'architecture : 1129
Surface reelle de l'architecture     : 2975
        sous16b_vhdl      : 1   (Surface : 52)
        mult16b_vhdl      : 1   (Surface : 1024)
         add16b_vhdl      : 1   (Surface : 53)
           Registres      : 20  (Surface : 70)
       Multiplexeurs      : 1   (Surface : 24)
     Demultiplexeurs      : 0   (Surface : 34)
             TriState     : 17  (Surface : 16)
                  Bus     : 3   (Surface : 50)

Limite technologique -> nombre d'ASIC : 2976

Utilisation des registres :
Taux moyen : 35.90%
```

```
Utilisation des operateurs:
        sous16b_vhdl 1 :  Rendement -->  2.56 %
        mult16b_vhdl 2 :  Rendement --> 87.18 %
         add16b_vhdl 3 :  Rendement --> 38.46 %
   Rendement moyen total --> 42.74 %

Memoire :    taille : 18  temps d'acces : 10
Seuil LOCAL - GLOBAL    : 40

Nombre optimal (taux) de Bus        : 1
Nombre optimal (taux) de Registres  : 7


**************************************************
Unite de controle

Nombre total d'etats en lineaire : 77 (7 bits)
Nombre total d'etats avec compteur : 39
Nombre total d'instructions : 39
Taille du registre d'etat : 6
Taille du compteur : 0

**************************************************

Ordonnancement des taches et Assignation sur operateurs

Operateur sous16b_vhdl 1 :
    Tache : 5 (sub), Tdeb : 0, Tfin : 10 (ASAP : 0 | ALAP : 0)
    Rendement -->   2.6 %
Operateur mult16b_vhdl 2 :
    Tache : 8 (mul), Tdeb : 20, Tfin : 40 (ASAP : 10 | ALAP : 10)
    Tache : 11 (mul), Tdeb : 40, Tfin : 60 (ASAP : 30 | ALAP : 30)
    Tache : 17 (mul), Tdeb : 60, Tfin : 80 (ASAP : 30 | ALAP : 30)
    Tache : 23 (mul), Tdeb : 80, Tfin : 100 (ASAP : 30 | ALAP : 40)
    Tache : 29 (mul), Tdeb : 100, Tfin : 120 (ASAP : 30 | ALAP : 50)
    Tache : 35 (mul), Tdeb : 120, Tfin : 140 (ASAP : 30 | ALAP : 60)
    Tache : 61 (mul), Tdeb : 140, Tfin : 160 (ASAP : 60 | ALAP : 60)
    Tache : 59 (mul), Tdeb : 160, Tfin : 180 (ASAP : 60 | ALAP : 60)
    Tache : 41 (mul), Tdeb : 180, Tfin : 200 (ASAP : 30 | ALAP : 70)
    Tache : 65 (mul), Tdeb : 200, Tfin : 220 (ASAP : 60 | ALAP : 70)
    Tache : 47 (mul), Tdeb : 220, Tfin : 240 (ASAP : 30 | ALAP : 80)
    Tache : 69 (mul), Tdeb : 240, Tfin : 260 (ASAP : 60 | ALAP : 80)
    Tache : 73 (mul), Tdeb : 260, Tfin : 280 (ASAP : 60 | ALAP : 90)
    Tache : 53 (mul), Tdeb : 280, Tfin : 300 (ASAP : 30 | ALAP : 90)
    Tache : 77 (mul), Tdeb : 300, Tfin : 320 (ASAP : 60 | ALAP : 100)
    Tache : 81 (mul), Tdeb : 320, Tfin : 340 (ASAP : 60 | ALAP : 110)
    Tache : 85 (mul), Tdeb : 340, Tfin : 360 (ASAP : 60 | ALAP : 120)
    Rendement -->  87.2 %
Operateur add16b_vhdl 3 :
    Tache : 14 (add), Tdeb : 60, Tfin : 70 (ASAP : 50 | ALAP : 50)
    Tache : 20 (add), Tdeb : 80, Tfin : 90 (ASAP : 50 | ALAP : 50)
    Tache : 26 (add), Tdeb : 100, Tfin : 110 (ASAP : 50 | ALAP : 60)
    Tache : 32 (add), Tdeb : 120, Tfin : 130 (ASAP : 50 | ALAP : 70)
    Tache : 38 (add), Tdeb : 140, Tfin : 150 (ASAP : 50 | ALAP : 80)
    Tache : 63 (add), Tdeb : 180, Tfin : 190 (ASAP : 80 | ALAP : 80)
    Tache : 44 (add), Tdeb : 200, Tfin : 210 (ASAP : 50 | ALAP : 90)
    Tache : 67 (add), Tdeb : 220, Tfin : 230 (ASAP : 90 | ALAP : 90)
    Tache : 50 (add), Tdeb : 240, Tfin : 250 (ASAP : 50 | ALAP : 100)
    Tache : 71 (add), Tdeb : 260, Tfin : 270 (ASAP : 100 | ALAP : 100)
    Tache : 75 (add), Tdeb : 280, Tfin : 290 (ASAP : 110 | ALAP : 110)
    Tache : 56 (add), Tdeb : 300, Tfin : 310 (ASAP : 50 | ALAP : 110)
    Tache : 79 (add), Tdeb : 320, Tfin : 330 (ASAP : 120 | ALAP : 120)
    Tache : 83 (add), Tdeb : 340, Tfin : 350 (ASAP : 130 | ALAP : 130)
    Tache : 87 (add), Tdeb : 360, Tfin : 370 (ASAP : 140 | ALAP : 140)
    Rendement -->  38.5 %
```

```
**************************************************

Assignation des registres

Registre 1 (0, 10) (Taux 2 %) :
   Operateur(s) en entree :   memoire
   Operateur(s) en sortie (operateur-patte) :   1-1
   Donnee yt (@yt) (LOC) : Tdeb : 0, Tfin : 10, entree : memoire, sortie : 1, patte
: 1
Registre 2 (0, 10) (Taux 2 %) :
   Operateur(s) en entree :   memoire
   Operateur(s) en sortie (operateur-patte) :   1-2
   Donnee y (@y0008) (LOC) : Tdeb : 0, Tfin : 10, entree : memoire, sortie : 1,
patte : 2
Registre 3 (20, 220) (Taux 51 %) :
   Operateur(s) en entree :   memoire
   Operateur(s) en sortie (operateur-patte) :   2-1
   Donnee deux_mu (@deux_mu) (LOC) : Tdeb : 20, Tfin : 40, entree : memoire, sortie
: 2, patte : 1
   Donnee x(0) (@x(0)) (LOC) : Tdeb : 40, Tfin : 60, entree : memoire, sortie : 2,
patte : 1
   Donnee x(2) (@x(1)) (VIE) : Tdeb : 60, Tfin : 220, entree : memoire, sortie : 2,
patte : 1
Registre 4 (10, 40) (Taux 7 %) :
   Operateur(s) en entree :   1
   Operateur(s) en sortie (operateur-patte) :   2-2
   Donnee lms8_minus_1 (GLO) : Tdeb : 10, Tfin : 40, entree : 1, sortie : 2, patte
: 2
Registre 5 (40, 300) (Taux 66 %) :
   Operateur(s) en entree :   2
   Operateur(s) en sortie (operateur-patte) :   2-2
   Donnee adapt (GLO) : Tdeb : 40, Tfin : 300, entree : 2, sortie : 2, patte : 2
Registre 6 (60, 310) (Taux 20 %) :
   Operateur(s) en entree :   memoire
   Operateur(s) en sortie (operateur-patte) :   3-1
   Donnee h(0) (@h(0)0001) (LOC) : Tdeb : 60, Tfin : 70, entree : memoire, sortie :
3, patte : 1
   Donnee h(1) (@h(1)0001) (LOC) : Tdeb : 80, Tfin : 90, entree : memoire, sortie :
3, patte : 1
   Donnee h(2) (@h(2)0001) (LOC) : Tdeb : 100, Tfin : 110, entree : memoire, sortie
: 3, patte : 1
   Donnee h(3) (@h(3)0001) (LOC) : Tdeb : 120, Tfin : 130, entree : memoire, sortie
: 3, patte : 1
   Donnee h(4) (@h(4)0001) (LOC) : Tdeb : 140, Tfin : 150, entree : memoire, sortie
: 3, patte : 1
   Donnee h(5) (@h(5)0001) (LOC) : Tdeb : 200, Tfin : 210, entree : memoire, sortie
: 3, patte : 1
   Donnee h(6) (@h(6)0001) (LOC) : Tdeb : 240, Tfin : 250, entree : memoire, sortie
: 3, patte : 1
   Donnee h(7) (@h(7)0001) (LOC) : Tdeb : 300, Tfin : 310, entree : memoire, sortie
: 3, patte : 1
Registre 7 (60, 370) (Taux 38 %) :
   Operateur(s) en entree :   2
   Operateur(s) en sortie (operateur-patte) :   3-2
   Donnee lms8_mult_2 (GLO) : Tdeb : 60, Tfin : 70, entree : 2, sortie : 3, patte :
2
   Donnee lms8_mult_3 (GLO) : Tdeb : 80, Tfin : 90, entree : 2, sortie : 3, patte :
2
   Donnee lms8_mult_4 (GLO) : Tdeb : 100, Tfin : 110, entree : 2, sortie : 3, patte
: 2
   Donnee lms8_mult_5 (GLO) : Tdeb : 120, Tfin : 130, entree : 2, sortie : 3, patte
: 2
   Donnee lms8_mult_6 (GLO) : Tdeb : 140, Tfin : 150, entree : 2, sortie : 3, patte
: 2
   Donnee y0001 (GLO) : Tdeb : 180, Tfin : 190, entree : 2, sortie : 3, patte : 2
   Donnee lms8_mult_7 (GLO) : Tdeb : 200, Tfin : 210, entree : 2, sortie : 3, patte
: 2
```

```
   Donnee lms8_mult_11 (GLO) : Tdeb : 220, Tfin : 230, entree : 2, sortie : 3,
patte : 2
   Donnee lms8_mult_8 (GLO) : Tdeb : 240, Tfin : 250, entree : 2, sortie : 3, patte
: 2
   Donnee lms8_mult_12 (GLO) : Tdeb : 260, Tfin : 270, entree : 2, sortie : 3,
patte : 2
   Donnee lms8_mult_13 (GLO) : Tdeb : 280, Tfin : 290, entree : 2, sortie : 3,
patte : 2
   Donnee lms8_mult_9 (GLO) : Tdeb : 300, Tfin : 310, entree : 2, sortie : 3, patte
: 2
   Donnee lms8_mult_14 (GLO) : Tdeb : 320, Tfin : 330, entree : 2, sortie : 3,
patte : 2
   Donnee lms8_mult_15 (GLO) : Tdeb : 340, Tfin : 350, entree : 2, sortie : 3,
patte : 2
   Donnee lms8_mult_16 (GLO) : Tdeb : 360, Tfin : 370, entree : 2, sortie : 3,
patte : 2
Registre 8 (70, 380) (Taux 23 %) :
   Operateur(s) en entree :   3
   Operateur(s) en sortie (operateur-patte) :   memoire
   Donnee h(0)0001 (@h(0)0001) (GLO) : Tdeb : 70, Tfin : 80, entree : 3, sortie :
memoire
   Donnee h(1)0001 (@h(1)0001) (GLO) : Tdeb : 90, Tfin : 100, entree : 3, sortie :
memoire
   Donnee h(2)0001 (@h(2)0001) (GLO) : Tdeb : 110, Tfin : 120, entree : 3, sortie :
memoire
   Donnee h(3)0001 (@h(3)0001) (GLO) : Tdeb : 130, Tfin : 140, entree : 3, sortie :
memoire
   Donnee h(4)0001 (@h(4)0001) (GLO) : Tdeb : 150, Tfin : 160, entree : 3, sortie :
memoire
   Donnee h(5)0001 (@h(5)0001) (GLO) : Tdeb : 210, Tfin : 220, entree : 3, sortie :
memoire
   Donnee h(6)0001 (@h(6)0001) (GLO) : Tdeb : 250, Tfin : 260, entree : 3, sortie :
memoire
   Donnee h(7)0001 (@h(7)0001) (GLO) : Tdeb : 310, Tfin : 320, entree : 3, sortie :
memoire
   Donnee y0008 (@y0008) (GLO) : Tdeb : 370, Tfin : 380, entree : 3, sortie :
memoire
Registre 9 (90, 320) (Taux 46 %) :
   Operateur(s) en entree :   3
   Operateur(s) en sortie (operateur-patte) :   2-2
   Donnee h(1)0001 (GLO) : Tdeb : 90, Tfin : 160, entree : 3, sortie : 2, patte : 2
   Donnee h(5)0001 (GLO) : Tdeb : 210, Tfin : 320, entree : 3, sortie : 2, patte :
2
Registre 10 (160, 190) (Taux 7 %) :
   Operateur(s) en entree :   2
   Operateur(s) en sortie (operateur-patte) :   3-1
   Donnee lms8_mult_10 (GLO) : Tdeb : 160, Tfin : 190, entree : 2, sortie : 3,
patte : 1
Registre 11 (190, 370) (Taux 46 %) :
   Operateur(s) en entree :   3
   Operateur(s) en sortie (operateur-patte) :   3-1
   Donnee y0002 (GLO) : Tdeb : 190, Tfin : 230, entree : 3, sortie : 3, patte : 1
   Donnee y0003 (GLO) : Tdeb : 230, Tfin : 270, entree : 3, sortie : 3, patte : 1
   Donnee y0004 (GLO) : Tdeb : 270, Tfin : 290, entree : 3, sortie : 3, patte : 1
   Donnee y0005 (GLO) : Tdeb : 290, Tfin : 330, entree : 3, sortie : 3, patte : 1
   Donnee y0006 (GLO) : Tdeb : 330, Tfin : 350, entree : 3, sortie : 3, patte : 1
   Donnee y0007 (GLO) : Tdeb : 350, Tfin : 370, entree : 3, sortie : 3, patte : 1
Registre 12 (50, 360) (Taux 74 %) :
   Operateur(s) en entree :   memoire  memoire
   Operateur(s) en sortie (operateur-patte) :   2-1
   Donnee x(1) (@xt) (VIE) : Tdeb : 50, Tfin : 160, entree : memoire, sortie : 2,
patte : 1
   Donnee xt (@xt) (LOC) : Tdeb : 160, Tfin : 180, entree : memoire, sortie : 2,
patte : 1
   Donnee x(7) (@x(6)) (VIE) : Tdeb : 200, Tfin : 360, entree : memoire, sortie :
2, patte : 1
Registre 13 (80, 260) (Taux 46 %) :
   Operateur(s) en entree :   memoire
```

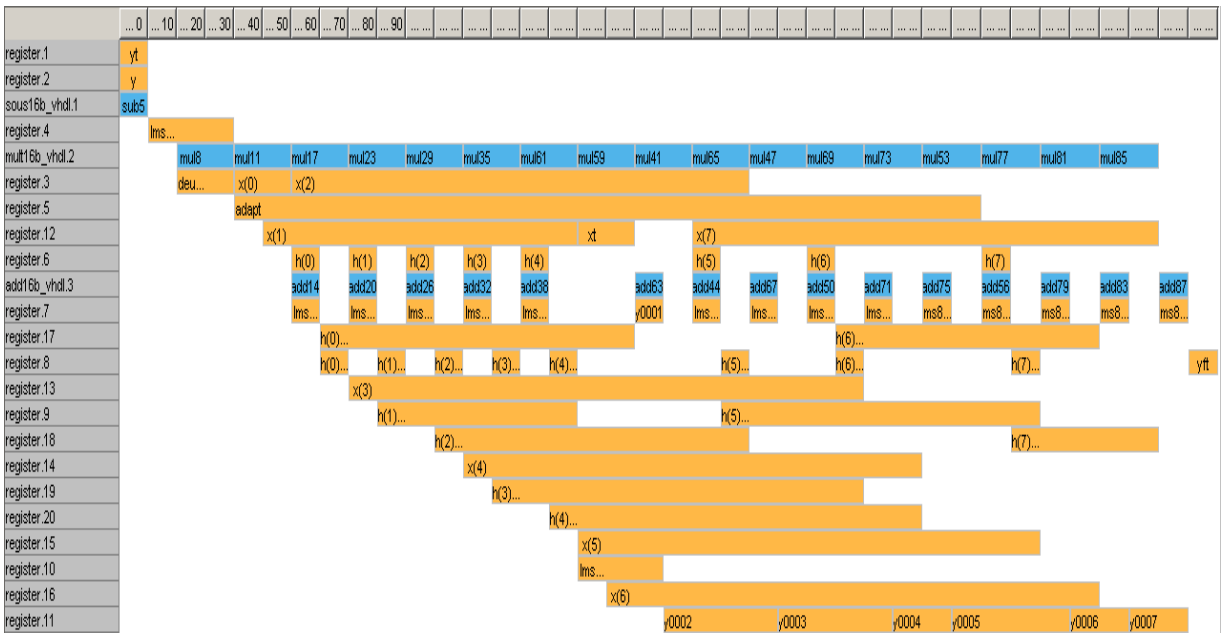
```
   Operateur(s) en sortie (operateur-patte) :   2-1
   Donnee x(3) (@x(2)) (VIE) : Tdeb : 80, Tfin : 260, entree : memoire, sortie : 2,
patte : 1
Registre 14 (120, 280) (Taux 41 %) :
   Operateur(s) en entree :   memoire
   Operateur(s) en sortie (operateur-patte) :   2-1
   Donnee x(4) (@x(3)) (VIE) : Tdeb : 120, Tfin : 280, entree : memoire, sortie :
2, patte : 1
Registre 15 (160, 320) (Taux 41 %) :
   Operateur(s) en entree :   memoire
   Operateur(s) en sortie (operateur-patte) :   2-1
   Donnee x(5) (@x(4)) (VIE) : Tdeb : 160, Tfin : 320, entree : memoire, sortie :
2, patte : 1
Registre 16 (170, 340) (Taux 43 %) :
   Operateur(s) en entree :   memoire
   Operateur(s) en sortie (operateur-patte) :   2-1
   Donnee x(6) (@x(5)) (VIE) : Tdeb : 170, Tfin : 340, entree : memoire, sortie :
2, patte : 1
Registre 17 (70, 340) (Taux 51 %) :
   Operateur(s) en entree :   3
   Operateur(s) en sortie (operateur-patte) :   2-2
   Donnee h(0)0001 (GLO) : Tdeb : 70, Tfin : 180, entree : 3, sortie : 2, patte : 2
   Donnee h(6)0001 (GLO) : Tdeb : 250, Tfin : 340, entree : 3, sortie : 2, patte :
2
Registre 18 (110, 360) (Taux 41 %) :
   Operateur(s) en entree :   3
   Operateur(s) en sortie (operateur-patte) :   2-2
   Donnee h(2)0001 (GLO) : Tdeb : 110, Tfin : 220, entree : 3, sortie : 2, patte :
2
   Donnee h(7)0001 (GLO) : Tdeb : 310, Tfin : 360, entree : 3, sortie : 2, patte :
2
Registre 19 (130, 260) (Taux 33 %) :
   Operateur(s) en entree :   3
   Operateur(s) en sortie (operateur-patte) :   2-2
   Donnee h(3)0001 (GLO) : Tdeb : 130, Tfin : 260, entree : 3, sortie : 2, patte :
2
Registre 20 (150, 280) (Taux 33 %) :
   Operateur(s) en entree :   3
   Operateur(s) en sortie (operateur-patte) :   2-2
   Donnee h(4)0001 (GLO) : Tdeb : 150, Tfin : 280, entree : 3, sortie : 2, patte :
2

**************************************************
Optimisation des bus externes
Nombre maxi de transfert simultane : 3
Nombre de ressource accedant a l'exterieur : 10

Cout supplementaire du a l'optimisation des bus : 24
```

File :

lms8.gantt : visualisation de l'ordonnancement des opérations et des données dans les registres



lms8_mem.gantt : visualisation de des accès aux mémoires

lms8_ES.gantt : visualisation de l'ordonnancement des entrées sorties