UNE NOUVELLE APPROCHE DE PLACEMENT DE DONNEES EN MEMOIRE : APPLICATION A LA CONCEPTION D'ARCHITECTURES D'ENTRELACEURS PARALLELES

Aroua Briki

THESE / UNIVERSITE DE BRETAGNE-SUD
sous le sceau de l'Université européenne de Bretagne
pour obtenir le titre de
DOCTEUR DE L'UNIVERSITE DE BRETAGNE-SUD
Mention :
Ecole doctorale SICMA

présentée par Aroua Briki
Préparée au Laboratoire Lab-STICC (UMR n°6285)
Université de Bretagne Sud

UNE NOUVELLE APPROCHE DE PLACEMENT DE DONNEES EN MEMOIRE : APPLICATION A LA CONCEPTION D'ARCHITECTURES D'ENTRELACEURS PARALLELES

Thèse soutenue le 01 juillet 2013
devant le jury composé de :

Amer Baghdadi (Telecom Bretagne / Lab-STICC)
Christophe Jego (ENSEIRB / IMS)
Smail Niar (Université de Valenciennes / LAMIH)
Adel Ghazel (Sup'Com Tunis)
Eric Martin (Université de Bretagne Sud / Lab-STICC)
Philippe Coussy (Université de Bretagne Sud / Lab-STICC)
Cyrille Chavet (Université de Bretagne Sud / Lab-STICC)

Résumé

Les applications du traitement du signal (TDSI) sont maintenant largement utilisées dans des domaines variés allant de l'automobile aux communications sans fils, en passant par les applications multimédias et les télécommunications. La complexité croissante des algorithmes implémentés et l'augmentation continue des volumes de données et des débits applicatifs requièrent souvent la conception de circuits intégrés dédiés (ASIC). Typiquement l'architecture d'un composant complexe du TDSI utilise (1) des éléments de calculs de plus en plus complexes, (2) des mémoires et des modules de brassage de données (entrelaceur/désentrelaceur pour les TurboCodes, blocs de redondance spatio-temporelle dans les systèmes OFDM[1]/MIMO, …). Aujourd'hui, la complexité et le coût de ces systèmes sont très élevés; les concepteurs doivent pourtant parvenir à minimiser la consommation et la surface total du circuit, tout en garantissant les performances temporelles requises. Sur cette problématique globale, nous nous intéressons à l'optimisation des architectures des modules de brassage de données (réseau d'interconnexion, contrôleur…) devant réaliser une règle d'entrelacement définie par l'application et ayant pour objectif d'utiliser un réseau d'interconnexion défini par le concepteur.

L'architecture que nous ciblons se compose d'éléments de calculs (PE_0,…PE_n), de mémoires de données utilisées pour stocker les opérandes et les résultats produits par les éléments de calculs (Mem_0,…Mem_m), d'un réseau d'interconnexion reliant les éléments de calculs aux mémoires et d'une unité de contrôle. Le réseau d'interconnexion est défini par l'utilisateur et peut être basé sur différent modèles : cross-bar, réseaux de Benes, réseau de Bruinj, barrière de multiplexeurs, barrel-shifters (barillets), papillons... L'unité de contrôle est composée de deux parties : un contrôleur de réseau et un contrôleur de mémoires. Ces contrôleurs sont basés sur un ensemble de mémoires de contrôle (une ROM de contrôle par banc mémoire *Mem* dans l'architecture cible) contenant les mots de commande relatifs au fonctionnement du système. L'approche que nous proposons est à même d'optimiser cette partie de contrôle de l'architecture.

Nous proposons plusieurs méthodologies d'exploration et de conception permettant de générer automatiquement une architecture d'entrelacement optimisée réalisant une règle de brassage de données, ou entrelacement, tel que définie par exemple dans un standard de communication.

Les approches que nous proposons prennent en entrée (1) des diagrammes temporels (générés à partir de la règle d'entrelacement et de contraintes spécifiant les séquences d'accès parallèles aux données) et (2) une contrainte utilisateur sur le réseau d'interconnexion que doit utiliser l'architecture. Ce flot formalise ensuite ces contraintes de brassage des données sous la forme (1) d'un modèle matriciel des séquences de données qui devront être traitées par chaque processeur et (2) d'un Graphe de Conflit d'Adressage (ACG), dont les propriétés permettent une exploration efficace de l'espace des solutions architecturales. L'objectif est ensuite de générer une architecture cible, en garantissant un fonctionnement sans conflit d'accès mémoire (lorsque plusieurs processeurs veulent accéder en même temps à un même banc mémoire mais pour traiter des données différentes), en respectant la contrainte de réseau et en optimisant l'architecture obtenue (notamment concernant l'architecture de son contrôleur).

Cette approche a été mise en œuvre au sein d'un d'outil et appliquée sur plusieurs cas d'étude : High Speed Packet Access (HSPA), Ultra-WideBand (UWB) et une application Wimax. Ces expériences montrent qu'en comparaison aux approches de l'état de l'art nos approches permettent d'atteindre des gains en surface significatifs. Notamment, pour des applications Turbo Codes pour lesquels les gains sont très importants.

[1]OFDM : technique de modulation se basant sur le multiplexage fréquentiel de signaux.

Abstract

Error correcting codes i.e. LDPC (Low Density Parity Check) and Turbo-codes are the foundation of communication. Standards like digital video broadcasting (DVB), high-speed wired links (ADSL…), wireless accesses (WiMAX, Wifi…), and telecommunications systems (HSPA, LTE…) all rely on it. LDPC and Turbo Codes are well-known, near Shannon limit, coding/decoding approaches that are able to achieve very low bit error rates for low Signal-to-Noise Ratio (SNR) applications. Decoding principle is based on message passing algorithm in which different processing elements iteratively exchange information in order to improve the error correction performance of these codes. In order to design high data rate applications, parallel architecture are needed. To increase memory bandwidth, main memory is divided into different memory banks to provide concurrent parallel access to all the processing elements. This allows to reduce the latency and thus to increase the throughput of decoders. Typical parallel decoder architecture includes processing elements connected through a dedicated Interconnection Network to memory banks and a dedicated Control Unit that drives the architecture. The network interleaves the data exchanged by the processing elements according to a rule named *interleaving law* or *permutation law* defined by the standard or the application to design. Unfortunately, depending on both interleaving law and memory mapping (i.e. data placement in memory banks), different processing elements may try to simultaneously access the same memory bank which results in memory conflicts.

Three kinds of solution exist to avoid or minimize memory access conflicts: (1) Define an interleaving law that automatically maps data in different memory banks so that all processing elements can access them without any conflict at each time instance, but only when the designer is free to choose the permutation law; (2) Simply store data elements in different memory banks without considering conflicting accesses and then use different complex topologies and additional buffers to mange conflicts on runtime. This increases the cost and latency of the system. (3) Use memory mapping algorithms to map data in different memory banks so that each processing elements can access them without any conflict. This kind of approach results in a non-optimized architectures. In addition, ROM resources are needed in the controller to store the memory mapping (i.e. control words to address memory banks and to drive interconnection networks). Unfortunately, cost of the controller is not considered for optimization in any state of the art approaches.

Our proposed approach is based on two main steps: first, starting from the set of constraints (i.e. the interleaving law, the parallelism and the targeted interconnection network), it generates a conflict free memory mapping (through a mapping algorithm based on memory constraint relaxation) and an *Address Conflict Graph ACG*. At this step, the data are assigned to memory banks (i.e. Bank Mapping), but their addresses in banks are still unknown (i.e. Address Mapping is not yet defined). In ACG, vertices represent data accesses in their respective memory banks, and edges represent conflicts between these accesses (e.g. write a new data at a currently used address). The memory mapping is performed thanks to a dedicated heuristic that aims to both generate a conflict free memory mapping and minimize the number of conflict in the generated ACGs. Indeed, when two data are stored in the same memory bank, and if their respective “lifetimes” in this memory bank are overlapping, two different memory locations (i.e. addresses) are required. Hence the bank mapping step strongly impacts the final ACGs, by generating more or less address conflicts. Then, memory mapping information and ACG are used during the *Address Mapping* step to explore the memory addressing in order to optimize the final cost of the memory controller. Finally, our design flow generates the final VHDL architecture with a conflict free memory mapping and the associated optimized control units.

Our approach has been applied to explore the design space of several test cases. The resulting architectures respect the designer architectural constraints in any case and the controllers are strongly optimized.

Table des matières

Table des figues

Table des tableaux

Chapitre 1

Introduction

Dans ce chapitre, nous présentons les techniques de codes correcteurs d’erreurs en mettant l’accent en particulier sur les Turbo-Codes et des codes LDPC. Les codes correcteurs d’erreurs peuvent être classés en deux catégories : les codes convolutifs et les codes en blocs. Nous présentons en premier lieu les techniques de codage et décodage dans les codes convolutifs. Nous introduisons ensuite brièvement les Turbo-Codes qui sont une sous classe des codes convolutifs. La dernière partie de ce chapitre est consacrée aux codes en blocs (plus particulièrement aux codes LDPC). Nous verrons que ces deux catégories présentent toutes les deux un même problème : elles sont sujettes aux problèmes de conflit d’accès aux mémoires. Finalement nous introduisons les problèmes de mise en œuvre de ces algorithmes dans une architecture parallèle afin de mettre en valeur l’intérêt des travaux que nous présentons.

I. Introduction

L'évolution des systèmes de communication amène leurs concepteurs à devoir traiter des volumes d'information en constante augmentation. Pour assurer à l'utilisateur une réactivité et une qualité de service acceptable, ces systèmes doivent garantir de très hauts débits applicatifs. Cette contrainte est aujourd'hui omniprésente dans les normes de télécommunication (DVB, 3GPP…)[DVBS08].

Afin d'atteindre les débits visés, ces normes imposent des Taux d'Erreurs Binaire (TEB) très faibles. Le nombre des erreurs de transmission dépend essentiellement du rapport signal à bruit du canal. Le bruit étant une contrainte physique irrémédiable, le contrôle algorithmique des erreurs de transmission devient indispensable. Ce contrôle se fait par la fonction de codage canal. La stratégie de base du codage consiste à rajouter de la redondance au message à transmettre. L'idée est d'affecter X messages numériques à C mots de codes, où $C>X$. On définit ainsi une notion de ratio ou taux ou rendement du code par $r = X/C$. Cette valeur qui indique la proportion de redondance introduite. L'ajout de la redondance en transmission permet au récepteur de détecter, localiser et corriger les erreurs de transmission plus efficacement. La Figure 1.1 modélise une chaine de communication typique.

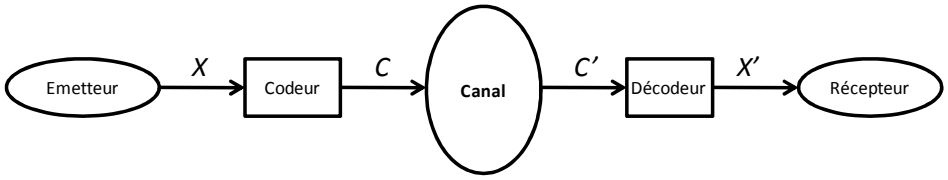


Figure 1.1 *Système de communication*

On distingue deux grandes familles de codes correcteurs d'erreurs selon la manière avec laquelle on ajoute la redondance : les codes en blocs (*Low Density Parity Check Codes, LDPC*) et les codes convolutifs (*Turbo-Codes*) [BER93]. Dans le premier cas, le message est d'abord divisé en blocs de données et le codeur traite ces blocs séparément. Par conséquent, le codeur doit attendre la réception d'un bloc pour démarrer. Dans le deuxième cas, le codeur traite le message de façon continue et génère séquentiellement les bits de redondance sans avoir besoin du message complet.

L'essentiel des standards de communication actuels se situent dans l'une de ces deux grandes familles de codes correcteurs d'erreurs (codes en blocs ou codes convolutifs). Néanmoins, exploiter ce type de codes, compte tenu des contraintes applicatives fortes que nous avons évoquées, n'est pas chose aisée et demande beaucoup d'expertise. A ce titre, parmi les problématiques résultantes de ces contraintes, l'une d'elles s'avère très importante compte tenu de l'impact qu'elle peut avoir sur le coût final de l'architecture : le parallélisme de traitement. En effet, les concepteurs doivent parvenir à construire des architectures capables d'atteindre les performances applicatives voulues (débit, taux d'erreur…) à moindre coût (temps de conception, surface matérielle, consommation…). Pour ce faire, ils peuvent ajuster le parallélisme de traitement de leurs architectures, malheureusement comme nous allons le voir, une architecture parallèle implique de nombreuses complications afin de s'assurer qu'aucun conflit d'accès aux données ne vienne perturber le bon fonctionnement du système. Dans cette thèse, les apports théoriques et méthodologiques que nous proposons ont pour objectif de répondre à ce défit tout en s'adaptant à la fois aux codes LDPC et aux Turbo-Codes.

A. Codes Convolutifs

Le fonctionnement des codes convolutifs est semblable à une machine à état fini qui traduit un flux continu de X bits d'information en C bits codés (avec C>X). Ils sont appliqués dans plusieurs standards de communication grâce à leurs structures simples et à la relative rapidité de leur implémentation. On retrouve ces codes convolutifs dans de nombreux standards de communication

mobiles (HSPA [HSP04], LTE [LTE08]) ainsi que pour la radiodiffusion mobile (DVB-SH [DVBS08]).

a) Les codeurs convolutifs

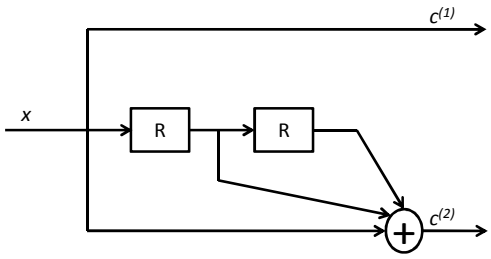
Les codeurs convolutifs se décomposent en opérateurs de calculs (i.e. des additionneurs) et en registres à décalage dont le rôle est la mémorisation des entrées précédentes ainsi que l'encodage de l'état. Le mot de code dépend ainsi non seulement du message d'information à son entrée, mais aussi des états de l'encodeur (les bits de message qui ont déjà été codés) et qui sont stockés dans les registres à décalage. Chacun de ces registres à décalage contient un ou plusieurs registres introduisant un retard d'une unité de temps. On peut distinguer deux catégories de codes convolutifs, les codeurs systématiques et les codeurs récursifs :

Définition Codeur Systématique

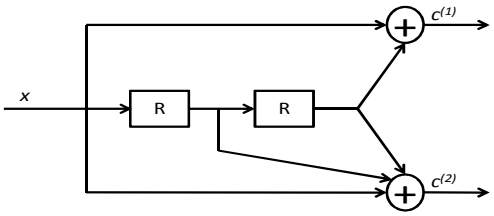
Un codeur convolutif est appelé *Systématique* si tous ses bits d'information d'entrée ont une correspondance directe avec les bits de sortie, sinon le codeur est dit *Non Systématique*.

Définition Codeur Récursif

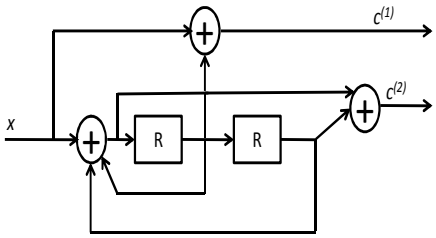
Un codeur convolutif est dit *Récursif* si son état dépend de ses sorties, sinon il est appelé *Non Récursif*.



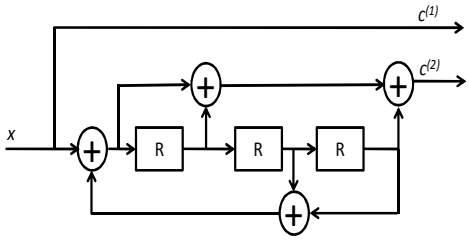
Codeur Convolutif systématique non-récursif



Codeur Convolutif non-systématique non-récursif



(c) *Codeur Convolutif non-systématique récursif*



Codeur Convolutif systématique récursif

Figure 1.2 *Codeur Convolutif*

D'après les définitions précédentes, il existe donc 4 types des codeurs convolutifs, illustrés dans la Figure 1.2a illustre un exemple de codeurs systématiques : Le bit d'entré x est directement lié au bit de sortie $c^{(1)}$. De même, des codeurs récursifs sont présentés dans les Figure 1.2.c et I.3.d.

- Codeur Convolutif systématique non récursif (Figure 1.2.a)
- Codeur Convolutif non systématique non récursif (Figure 1.2.b)
- Codeur Convolutif non systématique récursif (Figure 1.2.c)
- Codeur Convolutif systématique récursif (Figure 1.2.d)

Dans la Figure 1.2, les registres à décalages sont représentés par des carrées et les additionneurs modulo 2 sont représentés dans des cercles. Par la suite, afin d'expliquer les différentes terminologies utilisées dans les codes convolutifs, on considèrera le codeur illustré dans la Figure 1.2.d.

Le codeur systématique récursif est utilisé dans le standard *3GPP LTE*. A un instant donné *t*, le codeur génère à partir d'un bit d'information x_t deux bits codés $c_t^{(1)}$ & $c_t^{(2)}$, ce qui entraine un rendement de code 1/2. La concaténation des bits codés génère le mot code suivant $C = [c_1^{(1)}\ c_1^{(2)},\ c_2^{(1)}\ c_2^{(2)},\ c_3^{(1)}\ c_3^{(2)},\ldots,\ c_t^{(1)}\ c_t^{(2)}]$. Les bits de codes générés par les codeurs systématiques peuvent être différenciés selon deux types : les *bits systématiques* et les *bits de parité*.

Les bits d'information d'entrée sont les bits systématiques, en revanche les bits de sortie qui sont non systématiques sont appelés les bits de *parité*. Ainsi, dans la Figure 1.2d, les bits codés $c_t^{(1)}$ sont remplacés par les bits d'information $x^{(1)}$, et les bits codés $c_t^{(2)}$ sont remplacés par les bits de parité $p^{(1)}$, ceci génère le mot de code suivant :

$$C = [x_1^{(1)}\ p_1^{(1)},\ x_2^{(1)}\ p_2^{(1)},\ x_3^{(1)}\ p_3^{(1)},\ldots\ldots\ldots\ldots,\ x_t^{(1)}\ p_t^{(1)}]$$

b) Code convolutif et diagramme de treillis

Les codeurs convolutifs peuvent être modélisés par des machines à états finis. Dans un tel modèle, les relations entre les entrées, les états et les sorties peuvent être symbolisées via une table de transition d'état et d'un diagramme d'états.

Tableau1.1 : *Table de transition d'état*

Etat Courant S_c		Entrée $x_{c,n}$	Etat Suivant S_n		Sortie $c_{c,n}$	
S_c	$(s_c^{(1)} s_c^{(2)} s_c^{(3)})$	x	S_n	$(s_n^{(1)} s_n^{(2)} s_n^{(3)})$	$c_{c,n}^{(1)}$	$c_{c,n}^{(2)}$
S_0	000	0	S_0	000	0	0
S_0	000	1	S_4	100	1	1
S_1	001	0	S_4	100	0	0
S_1	001	1	S_0	000	1	1
S_2	010	0	S_5	101	0	1
S_2	010	1	S_1	001	1	0
S_3	011	0	S_1	001	0	1
S_3	011	1	S_5	101	1	0
S_4	100	0	S_2	010	0	1
S_4	100	1	S_6	110	1	0
S_5	101	0	S_6	110	0	1
S_5	101	1	S_2	010	1	0
S_6	110	0	S_7	111	0	0
S_6	110	1	S_3	011	1	1
S_7	111	0	S_3	011	0	0
S_7	111	1	S_7	111	1	1

Le codeur illustré dans la Figure 1.2.d contient trois registres qui sont les registres élémentaires formant le registre à décalage. Un état peut ainsi être représenté par $S = (s^{(1)}, s^{(2)}, s^{(3)})$, où $s^{(1)}, s^{(2)}, s^{(3)} \in \{0,1\}$ représentant le contenu des trois registres (un bit par registre) et est lu de gauche à droite. En effet, étant donné v le nombre des registres élémentaires, il existe 2^v états possibles pour le codeur.

Ainsi avec trois registres, le codeur peut avoir 8 états possibles: S_0 = (000), S_1 = (001), S_2 = (010), S_3 = (011), S_4 = (100), S_5 = (101), S_6 = (110), S_7 = (111). La table de transition d'état est présentée dans le Tableau1.1. Dans ce tableau, S_c est l'état courant, S_n est l'état suivant et x est le bit d'information qui cause la transition.

La table de transition d'état est représentée graphiquement par un diagramme d'état dont les nœuds représentent les états et les arcs représentent les transitions d'états. La Figure 1.3 illustre un exemple de diagramme d'état. L'étiquette présentée sur chaque arc mentionne le bit d'entrée qui génère la transition d'état ainsi que les bits de sortie (noté *Bit d'entrée / Bit de sortie*).

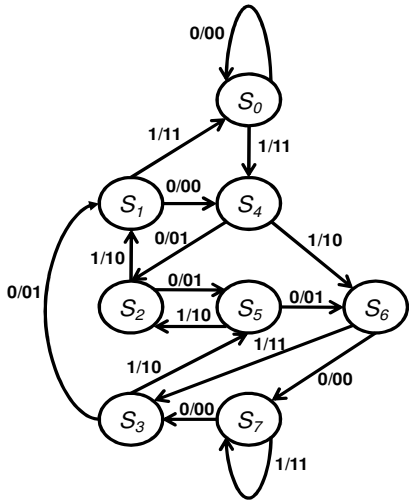


Figure 1.3 *Diagramme d'états*

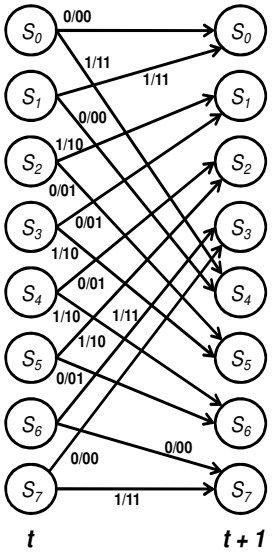


Figure 1.4 *Diagramme de treillis*

Le diagramme d'état montre la relation entre les transitions des états et les bits d'entrée/sortie, néanmoins il ne fournit aucune information concernant l'évolution temporelle. C'est le diagramme de treillis qui fournit cette information en étalant le diagramme d'état dans le temps (cf. Figure 1.4) : deux « copies » des états sont représentés aux instants t et $t+1$. Ces états sont mis en relation par des arcs orientés à partir des états à l'instant t vers les états à l'instant $t+1$; ces arcs permettent de représenter les transitions entre états.

Cette représentation sous forme de diagrammes de treillis, permet aux codes convolutifs d'être décodés de manière très fiable. En effet, il existe des algorithmes de décodage très efficaces opérant sur les codes de treillis ; chaque arc orienté dans le treillis correspond à un bit du mot de code, l'objectif de ces algorithmes consiste ainsi à déterminer à chaque instant l'arc orienté le plus probable correspondant au bit du mot de code le plus probable. Parmi les algorithmes de décodage utilisant des treillis, on trouve par exemple l'algorithme BCJR [BAH74] que nous détaillerons dans la section suivante.

c) Décodage des codes convolutifs

Si l'on s'en tient à ce que nous venons de voir, les transitions d'un état de S_c à un état S_n, dépendent de l'état courant et des entrées. Ces transitions peuvent se représenter sous la forme d'arcs dans un modèle de graphe dit de « treillis ». Le décodeur doit calculer la probabilité de transiter d'un état à un autre afin de trouver le bit d'entrée le plus probable.

L'algorithme de décodage sur treillis *BCJR* a été présenté par Bahl, Cocke, Jelenik et Raviv [BAH74]. Le principe de fonctionnement de celui-ci est basé sur le fait que les bits du mot de code c_t transmis à l'instant t sont influencés par les bits du mot de code c_t^- transmis avant l'instant t. Ainsi, il

est aussi possible que ces bits influencent les bits du mot de code c_t^+ transmis après l'instant *t*. Donc, afin d'estimer les bits du message reçu, l'algorithme effectue deux passages dans le diagramme de treillis : un passage vers l'avant (forward) et un autre dans vers l'arrière (bakward). Le premier se base sur les bits c_t^+ pour estimer les bits du message à l'instant *t*, le second lui utilise les bits c_t^-.

Soient,

Y_t : Les symboles reçus et qui correspondent aux bits du mot de code c_t transmis à l'instant *t*

Y_t^+: Les symboles reçus et qui correspondent aux bits du mot de code c_t^+ transmis après l'instant *t*

Y_t^- : Les symboles reçus et qui correspondent aux bits du mot de code c_t^- transmis avant l'instant *t*

La probabilité de la transition de l'état S_c à l'instant *t-1* à l'état S_n à l'instant *t* est calculée à partir de l'équation suivante :

$$\sigma_t(S_c, S_n) = A_{t-1}(S_c)\, \Lambda_t(S_c, S_n)\, B_t(S_n) \tag{1.1}$$

Cette probabilité est une fonction de différentes métriques :

(I) $A_{t-1}(Sc)$ représente la probabilité que le codeur soit à l'état S_c à l'instant *t-1*, sachant l'information sur les symboles y_t^- reçus avant l'instant *t*.

(II) $B_t(S_n)$ représente la probabilité que le codeur soit à l'état S_n à l'instant *t*, sachant l'information sur les symboles y_t^+ reçus après l'instant *t*.

(III) $\Lambda_t(S_c, S_n)$ représente la probabilité de la transition de l'état S_c à l'état S_n sachant l'information sur les symboles y_t reçus à l'instant *t*.

Les calculs des valeurs *A* et *B* sont respectivement appelés « *forward / backward recursion* » du décodeur BCJR. Les valeurs sont calculées à partir des équations suivantes :

$$A_{t-1}(S_c) = \sum_{i=0}^{2^v-1} A_{t-2}(S_i)\, \Lambda_{t-1}(S_i, S_n) \tag{1.2}$$

$$B_t(S_n) = \sum_{i=0}^{2^v-1} B_{t+1}(S_i)\, \Lambda_{t+1}(S_c, S_i) \tag{1.3}$$

Les équations montrent que les valeurs de *A* et *B* peuvent être calculées de façon récursive ; à chaque itération des opérations de multiplications par des nombres réels sont mises en œuvre. Ceci a pour conséquence d'augmenter la complexité du décodeur et donc son coût en termes de mise en œuvre matérielle. Pour s'affranchir de cette complexité, l'algorithme original est reformulé dans le domaine logarithmique, dans ce cas les opérations de multiplications sont remplacées par des additions.

Soient α, β et Γ des variables qui représentent les métriques sous forme logarithmique de *A, B et* σ Les équations précédentes sont transformées dans le domaine logarithmique de la manière suivante :

$$\Gamma_t(S_c, S_n) = \alpha_{t-1}(S_c) + \gamma_t(S_c, S_n) + \beta_t(S_n) \tag{1.4}$$

$$\alpha_{t-1}(S_c) = \log \sum_i e^{\alpha_{t-2}(Si) + \gamma_{t-1}(Si,\, Sn)} \tag{1.5}$$

$$\beta_t(S_n) = \log \sum_i e^{\beta_{t+1}(Si) + \gamma_{t-1}(Sn, Si)} \quad (1.6)$$

La dérivation de ces équations n'est pas détaillée dans cette thèse, le lecteur intéressé pourra consulter la référence [JOH10] pour avoir plus de détails théoriques et pratiques sur le sujet.

B. Turbo-Codes

Les Turbo-Codes [BER93], grâce à leurs excellentes capacités de correction d'erreurs, sont considérés comme une partie fondamentale des normes de télécommunication actuelles telles que LTE [LTE08], HSPA [HSP04], DVB-S [DVBS08]. Le turbo-codeur est la concaténation parallèle de deux codeurs convolutifs séparés par un entrelaceur qui permute les séquences de bits afin de casser les relations de voisinage entre eux, ce qui permet d'accroitre les capacités de correction d'erreurs.

Le décodeur est constitué par la concaténation en série de deux décodeurs convolutifs qui sont séparés par deux entrelaceurs. Ces deux décodeurs partagent leurs informations de manière itérative afin de décoder le message reçu. Ce procédé itératif, introduit au niveau du décodage, permet d'obtenir des gains de performances considérables et de se rapprocher à la limite de Shannon.[SHA48]. En outre, grâce à la faible complexité des algorithmes du décodage itératif, l'implémentation matérielle du Turbo-décodeur est simple à mettre en œuvre. Nous expliquons en détails dans la section suivante les différents éléments qui constituent un turbo encodeur.

a) Turbo-Codeur

Un turbo-codeur est la concaténation de deux codeurs convolutifs : le premier encode les bits d'informations X dans l'ordre naturel pour générer les bits de parité $p^{(1)}$, tandis que le second encode les bits d'informations $\Pi(X)$ dans l'ordre entrelacé (le message original est introduit dans l'entrelaceur), pour générer les bits de parités $p^{(2)}$, comme c'est indiqué dans la Figure 1.5.

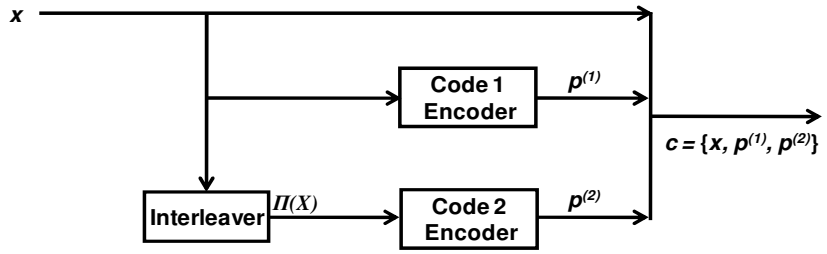


Figure 1.5 *Turbo-Codeur*

Les Turbo-Codes étant des codes systématiques, le mot de code à la sortie du codeur est construit par la concaténation parallèle des bits d'information X et des bits de parités générés par les deux codeurs ($p^{(1)}$ et $p^{(2)}$, cf. Figure 1.5). Les codeurs utilisés dans les Turbo-Codes sont généralement identiques, mais il est également possible d'utiliser des codeurs différents. Toutefois, du fait de la présence d'un entrelaceur, les bits de parité générés par les deux codeurs sont toujours différents même si les codeurs sont identiques.(cf.Figure 1.5).

b) Entrelaceur

L'entrelacement est exprimé à partir d'une séquence de permutation $\Pi = \{\pi_1, \pi_2, \pi_3, \ldots\ldots, \pi_n\}$ dont la séquence $\{\pi_1, \pi_2, \pi_3, \ldots\ldots, \pi_n\}$ représente la permutation des entiers de *1* jusqu'à *n.*

La fonction d'entrelacement consiste à générer deux bits de parités complètement différents une fois qu'ils sont introduits dans deux codeurs. La meilleure performance des Turbo-Codes est normalement réalisée par l'utilisation d'un entrelaceur qui effectue des permutations aléatoires pour des milliers de bits.

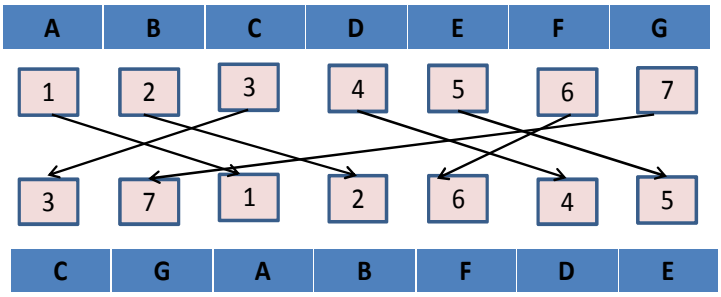


Figure 1.6 *Fonction d'entrelacement*

La Figure 1.6 montre un exemple de règle d'entrelacement représenté sous la forme d'une séquence de données *A, B, C, D, E, F, G* qui sera entrelacée selon la règle de permutation représentée par les séquences numériques. Le résultat de cette permutation sera le suivant : la séquence de données après permutation est la suivante *C, G, A, B, F, D, E.*

c) Turbo Décodeur

Le décodage de chaque code convolutif est réalisé par l'utilisation de l'algorithme BCJR [BAH74]: en turbo décodage, les deux décodeurs partagent l'information sur les bits du message reçu. Cette information est appelée information extrinsèque.

Chaque décodeur fournit l'information extrinsèque à l'autre décodeur afin d'estimer les bits de mot de code transmis par le codeur. Ensuite, dans le turbo décodage, l'information extrinsèque est mise à jour et partagée par les décodeurs durant plusieurs itérations. L'algorithme BCJR est donc appliqué plusieurs fois par chaque décodeur pour obtenir les bits du mot de code les plus probables. Le bloc diagramme du turbo décodeur est illustré dans la Figure 1.7 . Le décodeur reçoit des valeurs d'entrée $Y^{(u)}$, $Y^{(1)}$, $Y^{(2)}$ par le canal de transmission correspondant respectivement aux valeurs transmises par le codeur *X*, $p^{(1)}$, $p^{(2)}$, ces valeurs ont été définies dans la section 0.

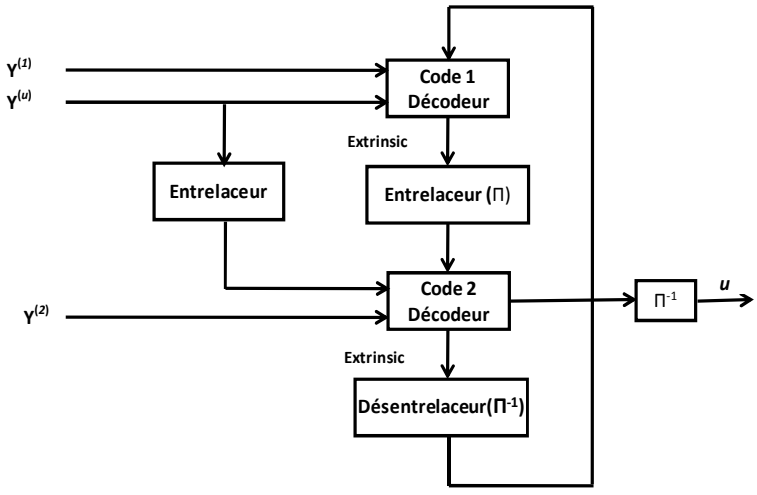


Figure 1.7 *Turbo-Décodeur*

Une itération complète du turbo décodeur est faite en deux demi itérations : le premier décodeur reçoit les valeurs par le canal correspondant aux bits du message $Y^{(u)}$, les bits de parité $Y^{(1)}$ et la valeur extrinsèque désentrelacée par le second décodeur afin de générer une nouvelle valeur extrinsèque. Cependant, dans la première itération le premier décodeur ne reçoit pas de valeur extrinsèque par le deuxième décodeur, il utilise uniquement $Y^{(u)}$, $Y^{(1)}$ pour produire la valeur extrinsèque. Durant la

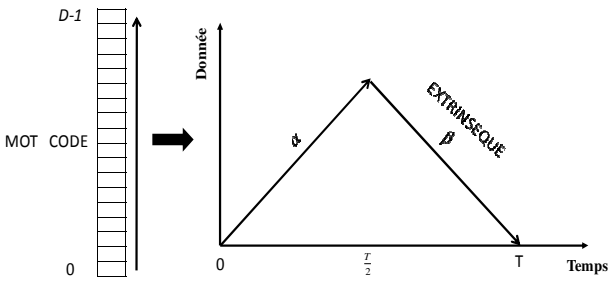
seconde demi itération, l'autre décodeur crée la valeur extrinsèque à partir des bits du message entrelacés, les bits de parité $Y^{(2)}$ et la valeur extrinsèque du premier décodeur.

Après un nombre fixé d'itérations, la décision finale sur les bits du message est basée sur les valeurs extrinsèques des deux décodeurs et les valeurs du canal. Il est important de noter que seules les valeurs extrinsèques sont mises à jour à chaque itération, les valeurs du canal correspondant aux bits du message restent toujours fixes.

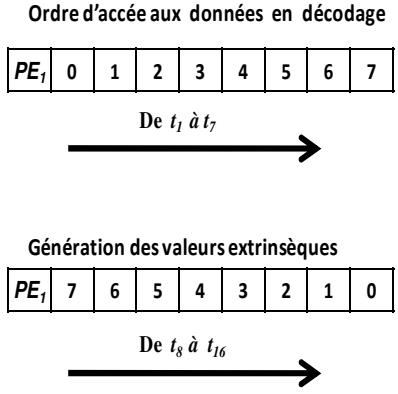
C. Parallélisme en Turbo Décodage

Comme décrit précédemment, afin de décoder chaque code, le turbo décodeur calcule en premier lieu les valeurs α en utilisant une approche dite «*Forward Recursion*», et les valeurs β en utilisant «*Backward Recursion*», ceci dans le but de générer les valeurs extrinsèques. On parle dans ce cas d'implémentation en *série* du turbo décodeur. Cette implémentation peut être représentée par un schéma «*Forward/Backward*», comme représenté dans la Figure 1.8 .a. L'implémentation de ce schéma produit la séquence d'accès aux données illustrée dans la Figure 1.8 .b.

Soient :
D = nombre des données utilisées pour un code = 8,
T = temps pour décoder un code 2D =16,
PE = nombre d'éléments de traitement =1,
Dans les premiers instants de 0 à $T/2 = 8$, le décodeur :
(1) accède à la mémoire extrinsèque afin de lire les données de $D = 0$ jusqu'à $D = 7$,
(2) calcule les valeurs α,
(3) les stocke dans la mémoire du décodeur.



(a) *Schéma "Forward/Backward"*



(b) *Ordre d'accès aux données pour une implémentation en série*

Figure 1.8 *Implémentation en série d'un Turbo décodeur*

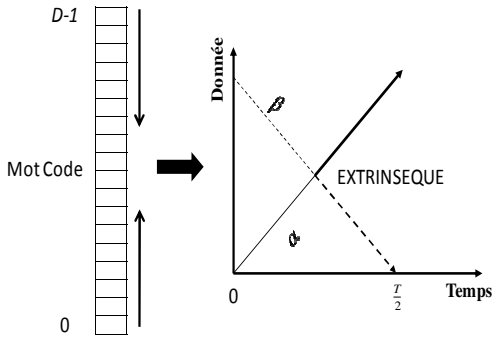
Puis de $T/2$ *à* T (de 9 jusqu'à 16), le décodeur calcule les valeurs de β et les valeurs extrinsèques, puis écrit dans la mémoire extrinsèque les valeurs extrinsèques actualisées. Il est important de noter que le décodeur peut traiter seulement une donnée à chaque instant dans l'implémentation en série (voir Figure 1.8).

Dans l'implémentation en série, le calcul des paramètres α et β ne peut se faire que lorsque toutes les données transmises ont été reçues et stockées, ce qui augmente le coût de la mémoire du décodeur. De fait, cette approche dégrade la latence globale du décodeur à cause du temps nécessaire pour le calcul des valeurs extrinsèques. Par conséquent, elle n'est pas réaliste si l'on souhaite cibler des applications à haut débits.

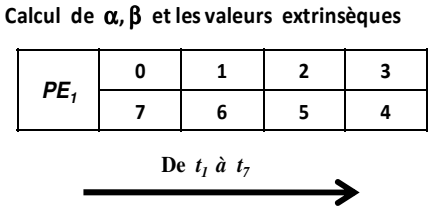
Afin de répondre à ces problématiques plusieurs approches, plusieurs techniques permettant d'accroître le parallélisme des architectures ont été proposées et sont décrites dans la partie suivante.

a) Parallélisme des unités récursives

Un premier type de parallélisme, dit « papillon » [ZHA04] est appliqué à l'algorithme de décodage en calculant en parallèle les valeurs α, β et les valeurs extrinsèques utilisées dans l'algorithme BCJR (cf. Figure 1.9 .a). Le décodeur calcule au même instant les paramètres α, β. Par conséquent, le schéma papillon augmente le degré de parallélisme en traitant deux données au même instant ; ainsi le temps nécessaire pour le décodage est divisé par deux. L'ordre d'accès aux données pour *D=8*, *PE =1* est indiqué dans la Figure 1.9 .b. Ce schéma parallélise l'unité récursive «*Forward/Backward*» en doublant le degré du parallélisme du processeur sans augmenter la surface de la mémoire du décodeur.



(a) *Schéma papillon*

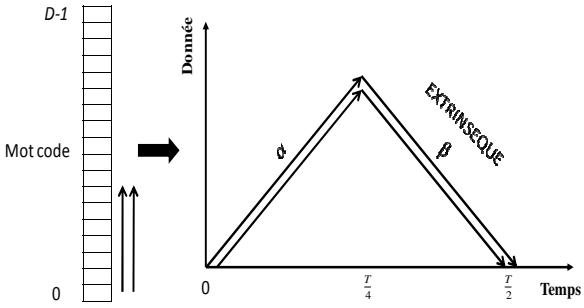


(b) *Ordre d'accès aux données dans le cas d'un traitement parallèle*

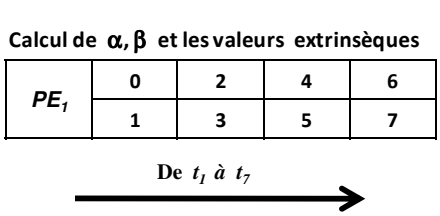
Figure 1.9 *Parallélisme de l'unité de calcul récursive*

b) Parallélisme au niveau du treillis

Dans le parallélisme au niveau du treillis [WOO00], au lieu de calculer pour chaque transition de treillis les paramètres α, β et les valeurs extrinsèques, ces valeurs sont calculées pour plusieurs transitions de treillis au même instant. Ceci a pour conséquence d'augmenter le degré de parallélisme qui est limité par le nombre total de transition dans un treillis. Le calcul «Forward/Backward» qui a été expliqué dans la section C est modifié pour ce type de parallélisme (cf. Figure 1.10 .a avec *S=2*). Le nombre des données traitées simultanément par chaque processeur est égal à *S*. L'ordre d'accès aux données pour *D=8*, *S=2* et *PE = 1* est illustré dans la Figure 1.10 .b.



(a) Schéma «Forward/Backward» modifié



(b) Ordre d'accès

Figure 1.10 *Parallélisme au niveau du treillis*

Grâce à ce type de parallélisme, les métriques utilisées au sein de l'algorithme BCJR sont réduites, il en résulte une diminution de la surface de la mémoire du décodeur. Par conséquent, cette approche permet ainsi de minimiser la quantité de ressource matérielles nécessaires [ASG10] car si les unités de calcul sont dupliquées, la mémoire du décodeur est elle réduite, compensant ainsi l'augmentation du coût de la surface des unités de calcul. En outre, le gain obtenu en termes de débit du décodeur comparé à la légère dégradation de sa surface a motivé les concepteurs pour exploiter ce niveau de parallélisme dans l'implémentation matérielles des turbo décodeurs [ASG10].

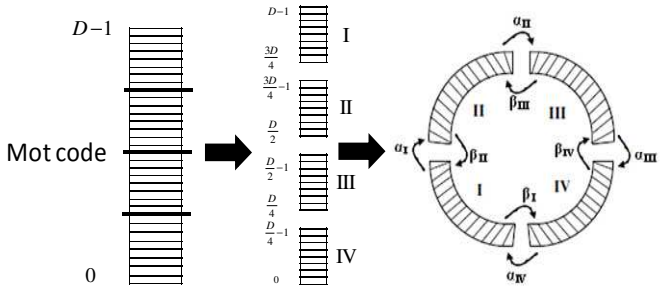
c) Parallélisme au niveau du décodeur SISO

Comme expliqué précédemment, l'implémentation en série d'un turbo décodeur peut se révélée très coûteuse notamment lorsqu'une grande quantité d'information doit être traitée car dans ce cas il faut augmenter proportionnellement la quantité de mémoire de l'architecture et la latence du système. Afin d'aborder ce problème, l'algorithme «Sliding window BCJR» [BLA05] a été proposé.

Dans cette approche, le bloc d'information est partitionné en segments (ou fenêtres) dont chacun est un sous ensemble de longueur est D_w du bloc original. Chaque fenêtre est traitée comme un bloc d'information séparé et alloué à un décodeur BCJR-SISO dédié. Ces décodeurs SISO fonctionnent en parallèle de manière à ce que les valeurs α, β, γ et les valeurs extrinsèques d'une fenêtre soient calculées au même instant que celles des autres fenêtres.

Puisque la fenêtre peut commencer, ou finir, en n'importe quel point du bloc d'information, l'algorithme de décodage BCJR utilise des valeurs initiales approximatives pour le calcul initial des paramètres α et β. Pour faire des estimations suffisamment précises, une première méthode consiste à estimer l'initialisation du sous bloc grâce à un pré calcul sur le bloc voisin. Ce pré calcul se base sur l'algorithme «Forward/Backward recursion» qui s'exécute sur un ensemble de bits supplémentaires provenant des fenêtres adjacentes à la fenêtre concernée, c'est ce qu'on appelle l'initialisation par *acquisition*. Une seconde méthode a été aussi introduite, nommée initialisation par passage de message «Message Passing», qui profite du processus itératif du décodage pour réutiliser les métriques d'état générées sur les fenêtres adjacentes à l'issue d'une itération comme initialisation dans l'itération suivante.

La Figure 1.11.a illustre la méthode «Sliding Window» et la technique «Message Passing». Le nombre des données qui sont traitées à un instant donné dépend du nombre des partitions et de la fenêtre courante. Pour chaque processeur qui traite une fenêtre, l'ordre d'accès aux données pour $D=8$, $PE=4$, $D_w=D/P=2$ est illustré dans la Figure 1.11.b.



«Sliding Window» et «Message Passing»

PE_1	0	1
PE_2	2	3
PE_3	4	5
PE_4	6	7

De t_1 à t_4

Ordre d'accès aux données (décodeur SISO)

Figure 1.11 *Parallélisme au niveau décodeur SISO*

Les valeurs approximatives initiales des métriques qui sont calculées en utilisant la technique de «Sliding Window» causent une dégradation considérée comme négligeable au niveau des performances du turbo décodeur. Cependant, cette technique permet d'améliorer le débit et de réduire la surface de la mémoire du décodeur.

d) Conclusion

Le processus du décodage d'un turbo-code avec l'algorithme BCJR peut faire intervenir du parallélisme à trois niveaux différents qui peuvent être tous exploités pour améliorer le débit du décodeur. Il est également possible de combiner ces différentes techniques pour obtenir un meilleur compromis temps/surface. Par exemple, on peut utiliser dans la même implémentation un parallélisme au niveau du treillis et un autre au niveau du décodeur SISO, la combinaison de ses deux techniques améliore le débit au prix d'un surcoût matériels. Néanmoins, l'implémentation de tous ces parallélismes cause des problèmes de conflits d'accès à la mémoire que nous détaillerons dans la section IV.

Dans le cadre de cette thèse, nous présentons des algorithmes permettant la résolution des problèmes de conflits mémoire et améliorant le coût de l'architecture générée quelque soit l'approche retenu par le concepteur (parmi celles que nous venons de voir).

II. Les codes en bloc

Les codes en bloc font partie de la seconde classe de codes correcteurs d'erreurs qui sont utilisés pour transmettre des données numériques de façon fiable via des canaux de communication non fiables, en présence de bruit. Plusieurs types de codes en blocs sont utilisés dans différentes applications. Parmi les codes de blocs classiques on peut citer : le Reed-Solomon [AHA] que l'on retrouve dans les CD, DVD et disques durs, les code Golay [GOL61] ou encore les codes de Hamming [HAM50].

Dans le codage en bloc, le flux de données est réparti en segments, ou blocs, de bits. Chaque bloc de message noté X contient x bits d'information : il en résulte 2^x mots de codes possibles. La fonction de codage consiste à ajouter de la redondance à un message d'information X afin de générer un mot de code C de longueur c bits avec $c>x$. Pour utiliser ce code en bloc dans la pratique, il est nécessaire que les 2^x mots de codes soient différents. Transformer le code en bloc en 2^x mots de codes de longueur c est une opération coûteuse en termes de surface puisque le codeur doit stocker 2^x mots de codes dans la mémoire. Afin de réduire cette complexité, les applications utilisent dans la pratique les codes en blocs linéaires.

Définition Code en bloc linéaire

> Un code en bloc linéaire est une classe de codes en blocs dans lesquels la somme de deux mots de code modulo 2 est également un mot de code.

Dans les codes en blocs linéaires, les mots de codes sont générés par une matrice génératrice G. Cette matrice contient x mots de codes de longueur c et qui sont linéairement indépendants. Le message d'information est multiplié par G pour générer le mot de code correspondant. L'exemple suivant explique la construction d'un code linéaire (7, 4) dont $c=7$ et $x=4$. Cet exemple ainsi que les informations sur les codes blocs linéaires sont extraits de [LIN04].

$$G = \begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Si X = (1 1 0 1) est le message d'information qui doit être codé alors son mot de code correspondant est :

C = X.*G*
C = **1**(*1 1 0 1 0 0 0*) ⊕ **1**(*0 1 1 0 1 0 0*) ⊕ **0**(*1 1 1 0 0 1 0*) ⊕ **1**(*1 0 1 0 0 0 1*)
C = (1 1 0 1 0 0 0) ⊕ (0 1 1 0 1 0 0) ⊕ (1 0 1 0 0 0 1)
C = (0 0 0 1 1 0 1)

Puisque le code en bloc linaire est spécifié par *G*, le codeur doit stocker les *x* lignes de *G* afin de pouvoir générer un mot de code de longueur *c* pour tout message d'information.

Tableau1.2 *Code en bloc linéaire avec x =4 et c =7*

Message	Mot de code
(0000)	(0000000)
(1000)	(1101000)
(0100)	(0110100)
(1100)	(1011100)
(0010)	(1110010)
(1010)	(0011010)
(0110)	(1000110)
(1110)	(0101110)
(0001)	(1010001)
(1001)	(0111001)
(0101	(1100101)
(1101)	(0001101)
(0011)	(0100011)
(1011)	(1001011)
(0111)	(0010111)
(1111)	(1111111)

Le Tableau1.2 montre tous les mots de code générés pour des messages de longueur *x=4* et des mots de code de longueur *c=7*. Il existe plusieurs types de codes en blocs parmis lesquels nous citons les codes en blocs linéaires..

A. Codage des codes en blocs linéaires

Comme expliqué précédemment l'encodeur a besoin de mémoriser *x* lignes de longueur *c* pour encoder n'importe quel message en un code en blocs linéaire (*c, x*). L'implémentation du codeur peut être simplifiée en introduisant une structure systématique durant la construction des codes en blocs linéaires. Dans cette structure, le mot de code est divisé en deux parties : le message et les bits de parité.

La partie parité du mot de code	La partie message du mot de code
$c - x$ bits	x bits

Figure 1.1 *Format systématique d'un mot de code*

La Figure 1.1 montre la structure du mot de code. Le message contient x bits d'information et les bits de parité sont de longueur c-x et représentent la somme linéaire des bits d'information. Un bloc linéaire avec une structure systématique est appelé *code en bloc linéaire systématique*.

Le mot de code (*7, 4*) figuré dans le Tableau1.2 est un code en bloc linéaire systématique, les x bits situés à droite du mot sont les bits d'information. Un code systématique linéaire est défini par sa matrice génératrice qui est divisé en deux matrices d'ordre $x*x$ et $x*p$, où $p = c$-x et $x*x$ est la matrice d'identité. La matrice génératrice pour le code en bloc linéaire systématique (*7, 4*) est définie de la manière suivante :

Matrice p Matrice d'identité $x*x$

$$G=\left(\begin{array}{cccc:cccc} 1 & 1 & 0 & \vdots & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & \vdots & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & \vdots & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & \vdots & 0 & 0 & 0 & 1 \end{array}\right)$$

Les codes linéaires systématiques deviennent ainsi relativement simples à produire. Supposons $X = (x_0, x_1, x_2, x_3)$ le message d'information qui doit être codé et $C = (c_0, c_1, c_2, c_3, c_4, c_5, c_6)$ le mot de code résultant. On a ainsi :

$$C = X.\begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Il en résulte par la multiplication des deux matrices :

$c_6 = x_3$
$c_5 = x_2$
$c_4 = x_1$
$c_2 = x_1 \oplus x_2 \oplus x_3$
$c_1 = x_0 \oplus x_1 \oplus x_2$
$c_0 = x_0 \oplus x_2 \oplus x_3$

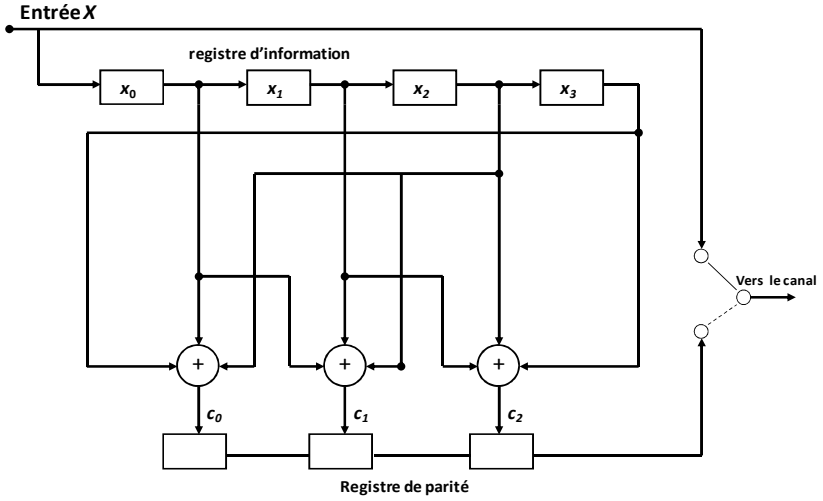


Figure 1.2 *Circuit du codage pour un code systématique(7,4)*

En exploitant ces équations, on peut générer un circuit de codage plutôt que de mémoriser les lignes de la matrice G [LIN04]. Le circuit de codage du code en bloc linéaire systématique *(7, 4)* est visualisé

dans la Figure 1.2. On retrouve sur ce schéma les 4 registres d'information $x_0 \ldots x_3$ qui sont utilisés pour calculer les bits de parités qui seront ajoutés au message originel c_0, c_1, c_2.

B. Codes LDPC "Low Density Parity Check Codes"

Les codes LDPC sont une classe des codes en blocs linéaires avec une capacité de correction d'erreur très proche de la limite théorique de Shannon[SHA48]. Grâce à leurs excellentes performances de correction d'erreurs, ils sont introduits dans plusieurs standard de communication sans fils tels que DVB-S2 et DVB-T2 [DVB08], WiFi (IEEE 802.11n) [WIF08] ou encore WiMAX(IEEE 802.16e) [WIM06].

Dans le cas des LDPC, la première étape de traitement consiste à modéliser le code avec les équations de vérification de parité. On considère le mot de code $C = [c_1\ c_2\ c_3\ c_4\ c_5\ c_6]$ qui satisfait les trois équations de vérification de parité suivantes :

$$c_2 \oplus c_3 \oplus c_4 = 0$$

$$c_1 \oplus c_2 \oplus c_4 = 0$$

$$c_1 \oplus c_3 \oplus c_4 = 0$$

Les contraintes du mot de code, ou équations de vérification de parité, sont modélisées par une matrice :

$$\underbrace{\begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}}_{H} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

La matrice de parité H ci-dessus est une matrice binaire $M*N$ dont chaque ligne M_i correspond à une équation de contrôle de parité tandis que chaque colonne N_j est associée à un bit de mot de code. Une valeur non nulle de cordonnée (i, j) dans la matrice H signifie que le $j^{ième}$ bit du mot de code est inclut dans la $i^{ième}$ équation de contrôle de parité. Un mot de code $x \in C$ est valide s'il vérifie l'équation suivante :

$$\forall x \in C, \quad xH^t = 0 \qquad (1.7)$$

Comme son nom l'indique, un code LDPC est un code en blocs qui contient une faible quantité de '*1*' dans la matrice de parité en comparaison avec la quantité de *0* dans la matrice « creuse ». Cette faible densité de la matrice H limite la complexité du décodage itératif, bien que celle-ci tende à augmenter linéairement avec la longueur du code. Le décodage des codes LDPC est basé sur la modélisation de l'équation 1.8 par un graphe biparti appelé graphe de Tanner.

C. Représentation du Graphe de Tanner

De part la faible densité de la matrice H, les codes LDPC peuvent être représentés par un graphe biparti appelé *Graphe de Tanner* qui associe un bit de code à une équation de contrôle de parité. Ce modèle de graphe est l'association de deux ensembles de sommets : les nœuds de variables (VNs) et les nœuds de parité (CNs). Une donnée $v_i \in VN$ représente un bit de mot de code (donnée à traiter) tandis que $c_j \in$ CN représente une équation de contrôle de parité utilisée pour la génération des bits de contrôle de parité (opération qui sera appliquée à la donnée). Un arc e_{ij} connecte le $i^{ième}$ nœud de parité au $j^{ième}$ nœud de variable (VN) si le $i^{ième}$ nœud de parité (CN) est contrôlé par le $j^{ième}$ nœud de variable. Cela signifie que le nombre d'arcs dans le graphe de Tanner est le même que le nombre de '*1*' dans la matrice H.

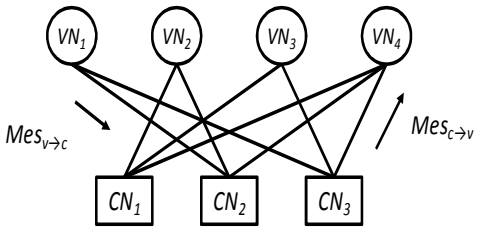


Figure 1.3 G*raphe de Tanner*

Le graphe de Tanner permet de comprendre le processus de décodage dont la fonction est d'échanger l'information entre CN et VN tout au long des arcs de ce graphe. Le graphe de Tanner de la matrice *H,* introduite dans la section précédente, est illustré dans la Figure 1.3 . Dans cette figure l'information allant de CN vers VN est représentée par $Mes_{c\to v}$ et l'information venant de VN et se dirigeant vers CN est représentée par $Mes_{v\to c}$.

D. Décodage

Comme expliqué dans la section précédente, les algorithmes de décodage des codes LDPC se basent sur des échanges d'information via les arcs du graphe de Tanner. Ces algorithmes sont appelés «Message Passing Algorithms», ce sont des algorithmes de décodage itératif dont les nœuds de variable et les nœuds de parité échangent de façon itérative l'information, jusqu'à ce que l'opération de décodage soit complète ou bien arrêtée selon des critères de performances définis par les concepteurs.

Selon le type des opérations à exécuter sur les nœuds, différents algorithmes sont disponibles, on peut citer par exemple : le décodage «Belief-Propagation», «Sum-Product» [PEA88], le décodage «Min-Sum» [FOS99] ou bien «Normalized Min-Sum» [CHE02].

L'algorithme «Sum-Product » est un algorithme d'échange d'information qui prend comme entrée des calculs de probabilités sur les valeurs des bits du mot de code. Afin d'avoir des calculs plus simples ses probabilités sont transformées dans le domaine logarithmique qui est appelé «Log Likelihood Ratio» (LLR). Ce LLR est défini par l'équation suivante :

$$LLR = \log\left(\frac{P(v=0)}{P(v=1)}\right) \qquad \textbf{(1. 8)}$$

avec $P(v = i)$ est la probabilité que v est égale à i.

Les entrées LLR sont appelés les valeurs «a priori» parce qu'elles sont connues avant même que le décodage des codes LDPC commence. Ce décodage se fait en trois étapes :

I- Initialisation

Les valeurs à priori de LLR sont assignées à tous les arcs sortants de chaque nœud de variable, le signe du LLR fournit une décision sur le bit transmis tandis que sa valeur absolue |*LLR*| donne une indication sur la fiabilité de cette décision.

II- Mise à jour du nœud parité

Dans cette étape, grâce à la $j^{ième}$ valeur CN, on estime la valeur du $i^{ième}$ VN en se basant sur les valeurs reçues par les autres VN connectés au même CN. Le CN crée une information extrinsèque supplémentaire sur la valeur du $i^{ième}$ VN. Chaque nœud de parité met à jour tous les nœuds de variables qui lui sont connectés en calculant et en transmettant ces informations extrinsèques. La loi de Bayes est utilisée dans le domaine logarithmique pour calculer le signe (équation 1.9) et la valeur absolue (équation 1.10) de l'information extrinsèque pour chaque nœud de variable.

$$sign(Mes_{c\to v}) = \prod_{v'\in v_c/v} sign(Mes_{v'\to c}) \quad (1.9)$$

$$\left|Mes_{c\to v}\right| = g\left(\sum_{v'\in v_c/v} g\left(\left|Mes_{v'\to c}\right|\right)\right) \quad (1.10)$$

Avec v_c représente l'ensemble de tous les *VN* qui sont connectés avec le *CN* courant, v_c/v représente tous les *VN* dans v_c sauf *v*. La fonction *g(x)* est calculée grâce à l'équation suivante *:*

$$g(x) = -\ln\tanh\left(\frac{x}{2}\right) = \ln\frac{\exp x + 1}{\exp x - 1} \quad (1.11)$$

III- La mise à jour du nœud variable

Dans cette étape, chaque VN met à jour ses valeurs en se basant sur l'information extrinsèque reçue par tous les CN qui lui sont connectés. La valeur de la sortie (SO), également appelée Probabilité A Postériori (APP), est calculée en utilisant l'équation 1.12, où le LLR est la valeur initiale d'entrée ou *valeur à priori*. La valeur SO est utilisée pour fournir au CN la nouvelle valeur de VN : $Mes_{v\to c}$. Ce message fourni au *CN* est calculé grâce à l'équation *1.13.*

$$SO_v = LLR + \sum_{c\in c_v} Mes_{c\to v} \quad (1.12)$$

$$Mes_{v\to c} = SO_v - Mes_{c\to v} \quad (1.13)$$

Processus itératif

Les processus de la mise à jour des nœuds de variable et de parité pour des nouvelles valeurs d'APP sont alors répétés jusqu'à ce que toutes les équations de parité soient satisfaites, ou bien qu'un nombre maximum d'itérations (défini par les concepteurs) soit atteint. A la fin du processus itératif (ou décodage), une décision « dure » est prise sur le mot de code de sortie en se basant sur les signes des valeurs de VN.

E. Implémentation du décodeur LDPC

L'implémentation matérielle d'un décodeur LDPC suppose de pouvoir réaliser la mise à jour des nœuds de variables (réalisation simple), et la mise à jour des nœuds de parité (réalisation complexe). En effet, la fonction *g(x)* utilisée dans la mise à jour des nœuds de parité est non linéaire. Néanmoins, afin d'avoir des résultats plus précis, un grand nombre de bits du mot de code est nécessaire, ce qui implique une augmentation du coût du système.

Pour réduire la complexité du calcul et le coût, des algorithmes sous-optimaux ont été proposés [FOS99] [CHE02], dans le but d'éviter l'évaluation de la fonction *g(x)*. L'algorithme « Min-Sum » simplifie l'algorithme « sum product » en remplaçant la fonction *g(x)* par le plus petit message entrant. Cette approche élimine la complexité de la mise à jour du nœud de parité et elle est exprimée de la manière suivante :

$$Mes_{c\to v}^{new} \approx \min_{v'\in v_c/v}\left|Mes_{v'\to c}\right| \quad (1.14)$$

La réduction de la complexité du calcul grâce à l'algorithme « Min-Sum » [FOS99] a toutefois pour conséquence la dégradation de la performance du décodage car l'approximation de la valeur absolue de l'information extrinsèque est toujours surestimée. Afin de limiter cette imprécision, un facteur de

normalisation μ est multiplié avec la sortie obtenue par l'équation (1.14). L'algorithme résultant est appelé « Normalized Min-Sum (NMS)» :

$$Mes_{c\to v}^{new} \approx \mu \min_{v'\in v_c/v} \left| Mes_{v'\to c} \right| \qquad (1.15)$$

avec μ facteur de normalisation $0 < \mu < 1$.

Malgré la simplification des calculs au niveau des nœuds du graphe de Tanner, le transfert des rdes informations via les arcs du graphe de Tanner n'est pas un processus simple, et il en résulte un nécessaire compromis entre un coût matériel acceptable et des contraintes de débit applicatif. Afin de faire face à ce problème de routage, trois types d'implémentations ont été introduits dans la littérature et sont décrits dans les sections suivantes.

a) Implémentation avec un parallélisme total

Dans cette architecture, chaque nœud et chaque arc d'un graphe de Tanner trouve une correspondance directe dans la réalisation matérielle du système via des circuits de calculs et des réseaux d'interconnexion dédiés. Les calcusl de tous les nœuds de parité et les nœuds de variable sont traités en deux étapes [BLA02] [FAN06] [NAG04] [ZHO07]. Grâce à cette architecture le décodage est extrêmement rapide, cependant le coût du circuit est important. En outre, la complexité et la rigidité d'une architecture d'interconnexion tirée d'un graphe de Tanner empêche une implémentation flexible des décodeurs LDPC.

b) Implémentation en série

Dans ce type d'architecture, les mises à jour des nœuds de variables et des nœuds de parité sont faites en série une par une. Cette architecture permet de réduire le cout matériel des décodeurs LDPC, cependant elle introduit une dégradation au niveau de la latence.

Par exemple, si T_{ch} cycles d'horloges par arc d'entrée sont nécessaires pour la mise à jour des nœuds de parité, et que T_{var} cycles d'horloges sont nécessaires par arc d'entrée, pour la mise à jour des nœuds de variables, alors l'implémentation en série exige $E_{Tan} * N_b(T_{ch} + T_{var})$ cycles d'horloge afin d'achever une seule itération de décodage LDPC. En outre, ce retard augmente proportionnellement avec l'augmentation de la longueur du code.

Par ailleurs, cette architecture exige une quantité de mémoire importante pour la mémorisation des messages transmis des nœuds de variables aux nœuds de parité, ainsi que pour les messages transmis des nœuds de parités aux nœuds de variables. Dans le cas d'un tel graphe de tanner, $2*E_{Tan} * N_b$ bits de surface mémoire sont nécessaires pour implémenter une architecture en série.

c) Implémentation avec un parallélisme partiel

L'implémentation avec un parallélisme partiel [MAS07] [LEE08] est un compromis entre un coût matériel excessif du fait d'une implémentation totalement parallèle et un faible débit causé par l'implémentation en série.

Dans cette architecture, plusieurs éléments de traitement (PEs) sont réalisés en matériel et ils sont partagés entre des groupes de nœuds de variable et de nœuds de parité. Afin d'atteindre des débits requis, un nombre approprié de processeurs doit être utilisé dans une telle implémentation. Les différents messages générés durant la mise à jour des nœuds de variables et des nœuds de parité sont stockés dans la mémoire. Ces messages sont écrits et lus selon une permutation particulière des arcs du graphe de Tanner.

Cette implémentation nécessite elle aussi de la mémoire, cependant cette architecture souffre de problèmes de collisions au niveau de l'accès à la mémoire, quand plus que deux processeurs veulent

accéder au même banc mémoire en même temps. Avec l'augmentation de la taille du mot de code, les conflits mémoires sont devenus une problématique importante et ils seront détaillés dans la prochaine section.

III. Les problèmes de conflits mémoires

Comme décrit précédemment, l'implémentation d'architectures parallèles de décodeurs LDPC ou de turbo-code suppose un compromis entre le coût du matériel et le débit. La Figure 1.1 montre une architecture typique d'une implémentation parallèle, dans cette figure, *P* processeurs (*PE*) dont le rôle est de traiter des données, communiquent avec *B* bancs mémoires (avec *P=B*) via un réseau d'interconnexion dédié. Un contrôleur se charge de piloter le fonctionnement du système (pilotage du réseau, contrôle des multiplexeurs, des accès mémoire…).

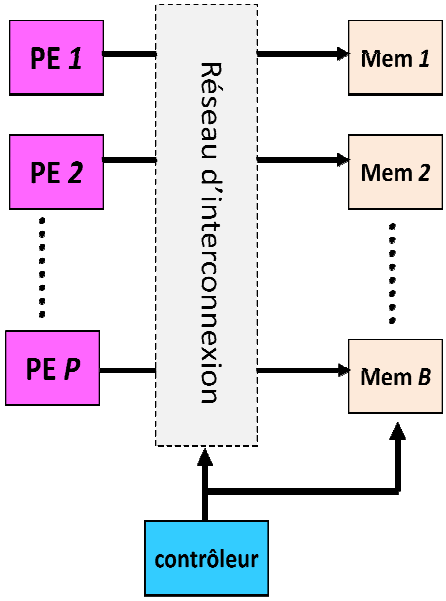


Figure 1.1 *Architecture partiellement parallèle*

Un conflit mémoires est présent si deux, ou plus, processeurs accèdent simultanément à des données distinctes stockées dans un même banc mémoire. Les conflits mémoires sont une problématique majeure dans la conception des architectures parallèles. Dans le domaine d'application que nous avons présenté, puisque l'accès en parallèle aux données peut se faire selon de nombreux schémas différents selon les algorithmes et les standards utilisés, nous présenterons dans un premier temps le problème séparément pour les Turbo-Codes, et les codes LDPC.

A. Les problèmes de conflits mémoire dans les Turbo-Codes

Comme expliqué précédemment, les entrelaceurs sont utilisés pour améliorer les performances de correction d'erreurs des Turbo-Codes. Ceci est obtenu en mélangeant les données de manière à ce que les bits de parités générés par les deux codeurs soient complètement différents.

Afin d'implémenter les Turbo-Codes parallèles présentés, les entrelaceurs doivent être également parallélisés, afin d'augmenter la bande passante de communication. Pour ce faire, la mémoire est divisée en blocs de mémoires plus petits, de telle sorte que plusieurs données puissent être extraites de la mémoire en même temps dans l'ordre naturel et que dans l'ordre entrelacé. De part le brassage des données qu'il induit, le parallélisme a pour conséquence d'augmenter le risque de conflit mémoire

(plusieurs données peuvent être accédées en lecture ou en écriture au même instant dans le même banc mémoire).

Nous présentons le problème à partir d'un exemple pédagogique d'entrelaceur. Soit un ensemble de *D=16* données. Nous structurerons cet ensemble sous la forme d'une matrice d'ordre *4*4*, les 4 premières données sont placées dans la première ligne de la matrice, les 4 suivantes dans la deuxième ligne … jusqu'à complétion de la matrice.

La Figure III.2.a montre la matrice d'entrelacement. Lorsque que les processeurs doivent accéder aux données en ordre naturel, on lit la matrice ligne par ligne, par contre, pour accéder aux données en ordre entrelacé, on lit la matrice colonne par colonne. Les deux ordres peuvent être présentés de la manière suivante :

Ordre naturel : *Contenu de la première ligne, contenu de la deuxième ligne,... contenu de la dernière ligne.*

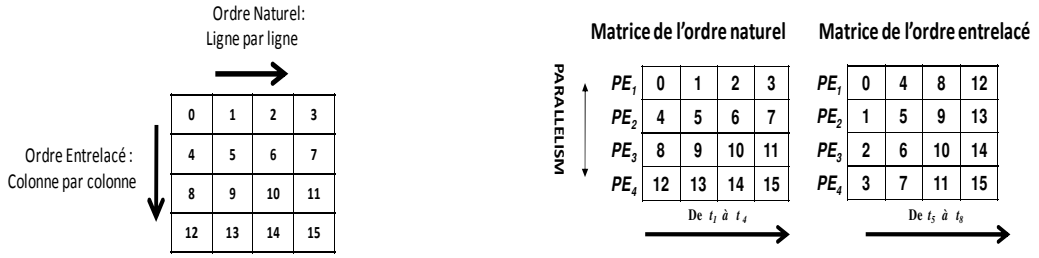
Ordre entrelacé : *Contenu de la première colonne, contenu de la deuxième colonne,… contenu de la dernière colonne.*

Appliqué à notre exemple on obtient :

Ordre naturel : *0,1,2,3 4,5,6,7 8,9,10,11 12,13,14,15*

Ordre entrelacé : *0,4,8,12 1,5,9,13 2,6,10,14 3,7,11,15*

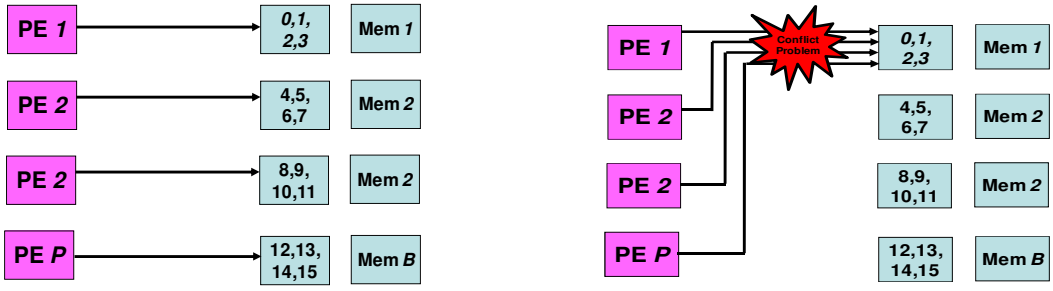
Dans un traitement parallèle utilisant la technique de "Sliding Window", le mot de code est divisé en 4 fenêtres dont chacune est traitée par un seul processeur, comme indiqué dans la Figure III.2.b



(a) *Entrelacement* ***(b)*** *Traitement parallèle*

Figure III.2 *Traitement parallèle pour les Turbo Codes*

Afin d'augmenter la bande passante de la mémoire, 4 bancs mémoires sont utilisés de manière à ce que les 4 processeurs puissent tous accéder en parallèle aux données. Le système doit alors être conçu de telle façon que les données nécessaires aux processeurs soient accessibles par ces derniers à chaque instant, à l'ordre naturel. Ceci suppose donc qu'à chaque instant les processeur accèdent tous à un banc mémoire distinct, comme indiqué dans la Figure 1.3.a.



(a) Accès à l'ordre naturel sans conflit (b) Accès à l'ordre entrelacé avec conflit

Figure 1.3 Les problèmes de conflits mémoire dans les turbo-décodeurs parallèles

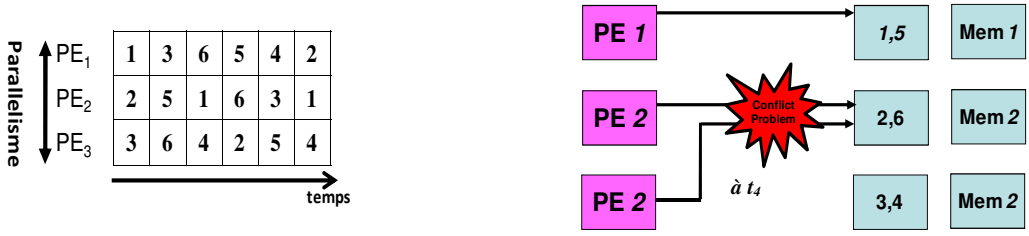
Toutefois, pour les accès en ordre entrelacé, on s'aperçoit que le placement des données en mémoire implique ici que tous les processeurs accèdent alors au même banc mémoire à un instant donné (cf. Figure 1.3.b). Ceci a pour conséquence de créer des conflits d'accès aux mémoires, et de dégrader la latence d'accès aux données dans la mémoire (à cause de la présence de mécanismes de gestion de ces conflits qu'il va falloir ajouter). Nous sommes là en présence d'un problème découlant directement du parallélisme de l'architecture.

B. Les problèmes de conflits mémoire dans les codes LDPC

Nous l'avons évoqué précédemment, si le calcul des nœuds de variables et des nœuds de parité est mathématiquement relativement simple, l'implantation matérielle d'une telle architecture pose des difficultés, notamment en terme de routage des nœuds variable et des nœuds parité. Il s'avère qu'une architecture partiellement parallèle est la solution la plus adéquate pour l'implémentation des décodeurs LDPC [MAS07][LEE08]. Malheureusement, cette architecture souffre elle aussi de problèmes de conflits d'accès aux mémoires.

Afin d'exposer le problème, nous introduisons une matrice d'affectation dans laquelle P lignes correspondant aux P processeurs, et N colonnes correspondant aux instants (ou cycles de calcul) t_i. Chaque colonne représente les données qui sont accédées en parallèle par les P processeurs à l'instant t_i. Les données situées dans une même ligne sont traitées par un même processeur.

La Figure 1.4.a représente la matrice d'affectation dans laquelle $D=6$ données, $P=B$ est le nombre des bancs mémoires, $M=D/B=2$ est la taille de chaque bancs mémoires et $N=6$ cycles de calculs.



Matrice d'affectation Matrice d'affectation

Figure 1.4 *Les problèmes de conflits mémoires dans les décodeurs LDPC partiellement parallèles*

Si les données stockées dans les bancs *Mem1*, banc *Mem2* et bancs *Mem3* sont respectivement (1, 5), (2, 6) et (3, 4) alors aux instants t_4 et t_5, on se retrouve dans une situation où plus d'un processeur veut accéder au même banc mémoire (cf. Figure 1.4.b). Ceci a, à nouveau, pour conséquence d'aggraver les problèmes de conflits mémoires, d'augmenter la latence ainsi que le coût du système.

IV. Problématique et contributions

Après avoir rappelé les principes de fonctionnement des turbo-codes et des LDPC, nous venons de montrer que le parallélisme nécessaire à l'implantation matérielle de tels codeurs/décodeurs, induit des problématiques nouvelles qui se retrouvent pour ces deux grandes familles applicatives. Ceci se traduit en particulier par la nécessité de résoudre les problématiques de conflits d'accès aux mémoires lors de la conception des entrelaceurs et du placement des données en mémoire.

Ces problèmes que l'on retrouve dans les Turbo-Codes aussi bien que dans les LDPC peuvent ralentir le processus du décodage et engendrer un surcoût en termes de ressources matérielles dût à la mise en œuvre des mécanismes de gestion des conflits. De ce fait, nous constatons que le coût de l'implémentation matérielle de ces entrelaceurs est également une problématique majeure qui ne doit pas être négligée par les concepteurs.

Résoudre les problèmes de conflits mémoire lors de la conception d'un entrelaceur parallèle permet d'assurer un fonctionnement cohérent de celui-ci et d'éviter de dégrader sa latence, néanmoins elle ne garantit pas la réalisation d'un système optimisé en terme de coût, de surface de silicium et de consommation. Les architectures parallèles sont toutefois une solution incontournable pour les applications de traitements du signal, en dépit du coût en termes d'éléments de mémorisation et de calcul ainsi que de complexité de contrôle. En outre, cette complexité est d'autant plus grande que le degré de parallélisme est grand. Nous présentons dans ce contexte un exemple d'architecture parallèle typique réalisant les Turbo-Codes et les codes LDPC (cf Figure 1.1).

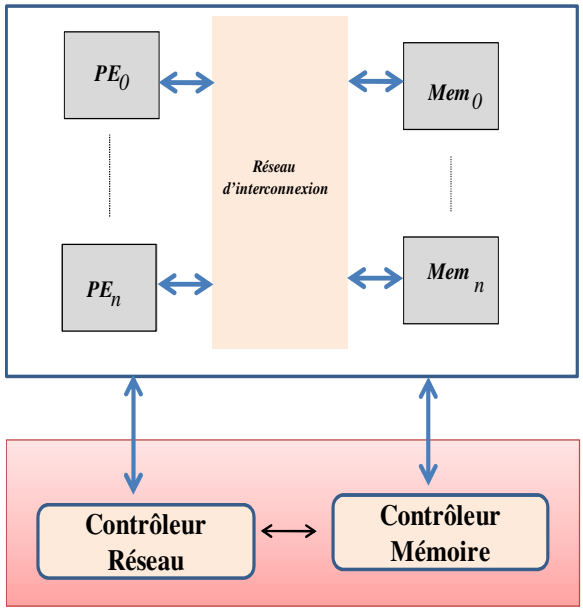


Figure 1.1 Architecture typique d'un entrelaceur parallèle

Le modèle d'architecture générique que nous ciblons se compose d'éléments de calcul (PE_0,…PE_n), de mémoires de données utilisées pour stocker les opérandes et les résultats utilisées par les éléments de calculs (Mem_0,…Mem_m), d'un réseau d'interconnexion reliant les éléments de calculs aux mémoires et un contrôleur pilotant le réseau d'interconnexion et les mémoires. Le contrôleur peut-être réalisé soit à l'aide de mémoires (type ROM) contenant les mots de commande relatifs au fonctionnement du système, soit à l'aide de machines à états (FSM - Finite State Machine). Le réseau d'interconnexion peut être basé sur différent types de réseau : cross bar, réseaux de Benes, réseau de Bruinj, barrière de multiplexeurs, barrel-shifters (barillets), papillons… Tous ceci constitue notre architecture cible, notre objectif étant d'en minimiser le coût, tout en garantissant un fonctionnant sans conflit mémoire.

Le coût de l'architecture tient pour une grande partie dans la complexité du contrôleur [SAN12], notamment si l'on veut concevoir des architectures multistandards. Celui-ci peut-être extrêmement complexe si l'on veut le construire à partir de machines à états finis génériques générant les mots de commande. A l'inverse, on peut déterminer statiquement (i.e. hors ligne) ces mots de commande et les mémoriser dans des mémoires ROM dédiées. Toutefois si le système doit être capable de gérer plusieurs modes de fonctionnement, le coût architectural sera là encore très élevé[SAN12]. Nous le verrons par la suite, les travaux de l'état de l'art ne répondent que partiellement ou imparfaitement à ces problèmes.

Nous proposons dans le cadre de cette thèse un flot de conception complet permettant de générer automatiquement des architectures d'entrelaceurs mémoire au niveau transfert de registres (RTL – Register Transfert Level) à partir d'une contrainte d'entrelacement donnée, par exemple définie dans

un standard de communication tel que ce que nous avons évoqués à plusieurs reprises. Notre architecture cible se compose des processeurs qui communiquent avec des mémoires à travers un réseau d'interconnexion et d'une partie contrôle. Nous introduisons dans ce flot de conception des modèles et des heuristiques capables de résoudre les problèmes de conflits mémoire, tout en optimisant le coût de l'implémentation matérielle de l'architecture générée. Plus particulièrement ces méthodes visent à réduire le coût matériel du réseau d'interconnexion ainsi que de la partie contrôle (contrôleur réseau et contrôleur mémoire).

V. Conclusion

Ce chapitre a introduit brièvement les deux grandes familles de codes correcteurs d'erreurs. Ces codes peuvent être divisés en deux catégories : les codes convolutifs et les codes en blocs, tous deux sont considérés comme les éléments clefs des standards de communication actuels.

Nous avons en premier lieu présenté dans ce chapitre les concepts utilisés dans les codages et décodages des codes convolutifs. Ces concepts sont utilisés pour expliquer les mécanismes à la base des Turbo-Codes (une sous classe des codes convolutifs) caractérisés par un taux d'erreurs binaires proche à la limite de Shannon. Nous avons également détaillé plusieurs techniques du décodage des Turbo-Codes pour des applications à haut débits. Puis, nous avons introduit une brève description des codes en blocs pour nous focaliser, sur le cas particulier des codes LDPC. Finalement, nous avons mis l'accent à travers des exemples sur les problèmes communs de conflits mémoires qui apparaissent lors de l'implémentation d'architectures parallèles pour ce type d'applications (Turbo-Code et LDPC).

Cette thèse explore des approches pour concevoir des entrelaceurs parallèles sans conflits mémoire et ayant un coût matériel réduit. Nous allons présenter dans le chapitre suivant un état de l'art sur les approches permettant d'implémenter les entrelaceurs parallèles. Celles-ci permettent de résoudre les problèmes de conflit mémoire, cependant elles génèrent souvent des architectures complexes et coûteuses.

Chapitre 2

Etat de l'art

Table des matières

Dans le chapitre précédent, nous avons introduit différentes techniques de correction d'erreurs permettant de réaliser une transmission fiable entre un émetteur et un récepteur. Nous avons également présenté différentes architectures de décodeurs permettant d'implémenter ces techniques. Les architectures parallèles sont nécessaires pour respecter les débits applicatifs et permettent de surcroit d'obtenir un bon compromis entre le coût architectural et les performances. Néanmoins, comme nous l'avons évoqué, ce type d'architecture souffre de problèmes de conflits d'accès aux mémoires. Dans ce chapitre, nous proposons une étude des solutions existantes dans la littérature permettant de répondre à ces problèmes inhérents aux architectures de codeurs/décodeurs parallèles.

I. Introduction

Fondamentalement, la problématique de placement de données en mémoires sans conflit se ramène à un problème de coloriage de graphe. Toutefois, comme nous allons le voir, les algorithmes existant dans le domaine de la théorie des graphes sont soit inefficaces, soit non pertinent dans la pratique s'ils sont utilisés tel quel [CHA10]. Toutefois avant de rentrer dans ces aspects théoriques, nous présentons les algorithmes et les définitions utilisées qui permettront de comprendre certaines approches utilisées pour concevoir des entrelaceurs sans conflits mémoires.

Définitions Graphe

Un graphe $G = (V, E)$ est un ensemble de nœuds V, et un ensemble d'arcs E.

Si $v,w \in V$ alors un arc $(v, w) \in E$ est incident à v et à w, et les sommets v and w sont dits adjacents.

Un sous graphe de G est un graphe dont les sommets et les arcs sont dans G.

De nombreux problèmes théoriques animent la communauté scientifique autour de la théorie des graphes. Notamment le problème de coloriage minimal des nœuds d'un graphe qui, comme nous le verrons par la suite, permet de formaliser le problème de conflit d'accès à des bancs mémoire.

Les graphes de conflits

Dans [KEY01], les problèmes d'allocation mémoires sont modélisés par des graphes de conflit. Dans un graphe de conflit, toutes les données sont modélisées par des nœuds. Deux données sont reliées par un arc si elles sont accédées en parallèles (i.e. en même temps). Incidemment, cela signifie que ces deux données doivent être stockées dans deux mémoires différentes.

Soient deux données accédées en parallèle, si l'on assimile les bancs mémoires aux couleurs, alors cela revient à dire que les deux données doivent être colorées avec deux couleurs différentes puisqu'elles sont adjacentes dans le graphe de conflit.

Définition Graphe de conflit

Un graphe de conflit $G(N,E)$ est un graphe dont l'ensemble de nœuds $N = \{n_1,n_2, \dots , n_n\}$ est défini de la manière suivante :

(a) $n_i \epsilon N$, le nœud n_i correspond à une donnée utilisée dans le calcul avec $i = \{1,2,\dots,n\}$.

(b) $e_{ij} \epsilon E$, l'arc e_{ij} est incident entre le nœud n_i et le nœud n_j si les données i et j sont conflictuelles.

Une fois le graphe de conflit construit, le problème d'allocation devient alors un problème de coloriage de graphe défini de la manière suivante :

Si on considère n couleurs, puisqu'on a n bancs mémoires qui sont accédés en parallèle, alors le problème de l'assignation des structures de données aux bancs mémoires, de telle sorte qu'aucun conflit ne se produit durant l'exécution de l'architecture parallèle, est un problème de coloriage de graphe.

Problème Problème de coloriage des nœuds

Etant donné un graphe *G*, est-il possible que les nœuds de *G* soient colorés avec *n* couleurs en respectant la condition suivante : les nœuds adjacents doivent avoir des couleurs différentes avec *n* le nombre minimal nécessaire de couleurs pour colorier les nœuds de *G*.

Nous le verrons par la suite, de nombreuses approches de l'état de l'art traitent les problèmes de conflits d'accès à des bancs mémoire en utilisant sous une forme ou une autre, de tels graphes de conflits. Dans [KOZ92] il a été prouvé que le coloriage des nœuds est un problème NP–complet, et nous verrons par ailleurs qu'utiliser des approches de résolutions basées sur des algorithmes de coloriage de graphes ne permet pas de répondre systématiquement au problème.

Dans une première partie, nous étudierons un ensemble d'approches consistant à construire une règle d'entrelacement qui soit intrinsèquement sans conflit autour d'un réseau d'interconnexion ciblé. Puis, nous introduirons des approches proposant de gérer les conflits d'accès aux mémoires directement dans le réseau d'interconnexion et à la volée, i.e. en cours d'exécution du système, via une architecture dédiée. Enfin, une dernière famille d'approches consiste à utiliser des algorithmes de placement de données en mémoire qui assignent les données dans des bancs mémoires de manière à ce que tous les processeurs puissent accéder, en parallèles, aux bancs mémoires sans aucun risque de conflit.

II. Approches de résolution des conflits mémoires pour Turbo-Codes et codes LDPC

A. Définition de règles d'entrelacement sans conflits

Nous décrivons ici un premier type d'approche consistant à définir une règle d'entrelacement nativement sans conflit. On trouve de telles méthodes à la fois pour les Turbo-Codes et les codes LDPC.

a) Turbo-Codes à roulettes

L'objectif des concepteurs est ici de définir des règles d'entrelacement sans conflit et garantissant de bonnes performances en termes de correction d'erreur. Ces règles d'entrelacement permettent un accès en parallèle de tous les processeurs à tous les bancs mémoires sans risque de conflit.

Dans [GNA03] une telle approche, appelée « Turbo codes à roulettes », est décrite. Elle est basée sur la définition d'une règle d'entrelacement circulaire permettant de réaliser un entrelaceur ayant un haut degré de parallélisme avec des performances équivalentes ou meilleures que celles des Turbo-Codes, tout en permettant de réduire la complexité de l'architecture d'interconnexion, en particulier pour des applications à très haut débit.

Les Turbo-Codes à roulettes sont construits à la manière suivante : La trame d'information de N symboles est découpée en P blocs de M symboles, avec $N=M.P$. Comme pour un Turbo-Codes classique l'opération de codage est d'abord effectuée dans l'ordre naturel afin de générer la redondance dans la première dimension. Chaque bloc est codé de manière indépendante par un codeur Convolutif Récursif Systématique Circulaire (CRSC). La trame d'information est ensuite permutée par un entrelaceur de taille N symboles. La trame entrelacée est également découpée en P blocs de longueur M et chaque bloc est codé indépendamment par un code CRSC pour produire la redondance de la deuxième dimension.

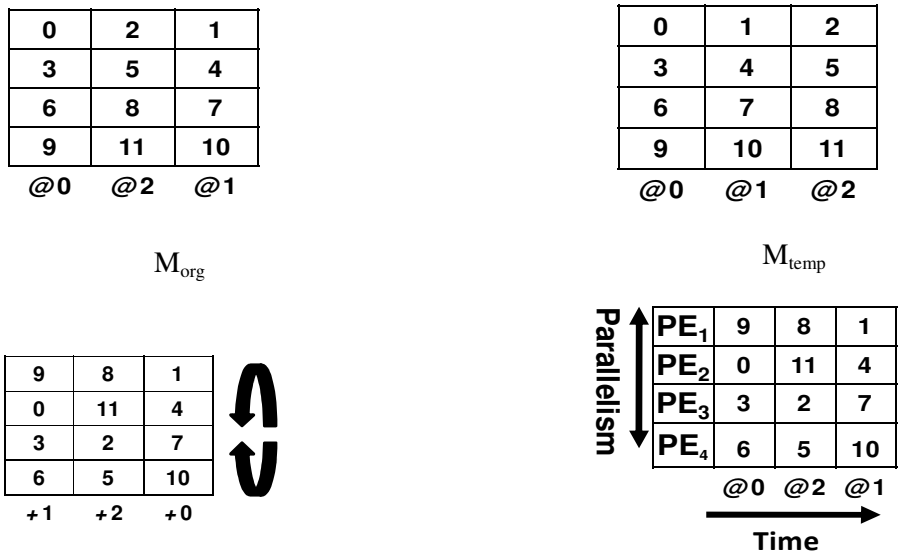


Figure 2.1. *Permutation spatiale et temporelle pour la construction d'un entrelaceur*

La méthodologie de conception de l'entrelaceur s'appuie sur une permutation spatiale et temporelle des données. Afin d'llustrer cette technique nous introduisons un simple exemple : considérons un bloc de longueur *12* symboles rangés ligne par ligne dans une matrice M_{org}, comme indiqué dans la Figure 2.1.a. La fonction de l'entrelaceur consiste à agréger les permutations aussi bien spatiales que temporelles. Dans une seconde étape, la permutation temporelle est obtenue en changeant les positions des colonnes dans la matrice M_{org} ce qui génère ainsi la matrice M_{temp} comme le montre la Figure 2.1.b. Finalement une permutation spatiale est réalisée en appliquant une permutation circulaire au niveau des différentes colonnes pour obtenir la matrice entrelacée M_{spa} comme s'est indiquée dans la Figure 2.1.c. Chaque ligne est traitée par un seul processeur et la taille de la mémoire est représentée par le nombre des colonnes comme le montre la Figure 2.1.d.

Cette approche permet d'utiliser un réseau d'interconnexion simplifié (e.g un « barrel shifter ») si le concepteur parvient à produire la règle d'entrelacement adéquate. Toutefois, cette technique est basée sur la mise en œuvre d'une règle d'entrelacement spécifique, donc non conforme à un standard, et ne peut pas être appliquée dans le cas d'autres règles d'entrelacement. Des techniques similaires ont été utilisées dans les documents [GIU02] et [KWA02] pour concevoir des entrelaceurs sans conflits.

b) Entrelaceurs déterministes

Certains des derniers standards de communication tendent quand à eux à définir dès leur conception des entrelaceurs sans conflit. On parle alors d'entrelaceurs déterministes. Toutefois plusieurs limites restent : le coût de l'architecture globale n'est pas pris en compte (ces approches ne permettent pas d'optimiser le coût des parties de contrôle) et le problème de conflits d'accès reste toujours pour l'entrelaceur canal.

Ces entrelaceurs déterministes ont deux inconvénients : d'une part les contraintes architecturales qui s'imposent durant la conception de ces entrelaceurs s'opposent à la performance de la correction d'erreur des turbo-codes. D'autre part, ils sont performants pour certains degrés de parallélismes ou certaines tailles de trames.

Selon ce principe on peut par exemple citer le standard 3GPP LTE [LTE08]. L'entrelaceur est réalisé par une permutation quadratique polynomiale (QPP) [SUN05]. Pour une trame de grande taille, la performance du décodage de l'entrelaceur QPP est proche de celle de l'entrelaceur « aléatoire »,

inversement pour les trames plus courtes, la performance de l'entrelaceur QPP est meilleur que celle de l'entrelaceur « aléatoire ». Il s'est avéré également [TAK06], que le risque d'avoir des conflits mémoires dans l'entrelaceur *QPP* est minimal, il est appelé « maximum contention-free interleaver».

c) Les codes LDPC structurés

Comme indiqué dans le chapitre précédent, un code LDPC est défini par sa matrice de parité *H*. Il est nécessaire que cette matrice ait des caractéristiques bien spécifiques pour obtenir de bonnes performances de correction d'erreurs. Différentes contraintes peuvent être ajoutées durant la construction de la matrice *H* dans le but d'atteindre des gains de codage significatifs ou bien pour simplifier l'architecture du décodeur. Dans un premier type d'approche, la matrice *H* est construite de telle sorte que le transfert des données entre les nœuds de variables et les nœuds de parité peut être fait sans conflit dans les architectures partiellement parallèles. Afin de faire face aux problèmes de conflit mémoire, la matrice *H* est divisée en différents blocs de sous matrices dont chacune est obtenue en permutant les lignes de la matrice identité [ZHA01] [MAN03].

Le code obtenu par l'ajout de cette régularité durant sa construction est appelé *code structuré*. Dans ce type de code les problèmes de conflits mémoire n'existent plus car le transfert de l'information entre les nœuds variables et les nœuds de parité sont effectués en utilisant des règles simples. De plus, les codes structurés simplifient l'architecture du décodeur car le réseau d'interconnexion peut être simple (exemple « barrel shifter ») en exploitant la régularité introduite durant la construction du code.

Grâce à leur simplicité de construction, les codes structurés font partie de standards de télécommunications courants telles que IEEE 802.11n (WiFi) [WIF08] et IEEE 802.16e (WiMAX) [WIM06]. Dans ces standards, les codes structurés sont construits en divisant la matrice originelle en différentes sous-matrices comme s'est indiqué dans la Figure 2.**2**.a. Cette matrice est constituée de *X* lignes et *Y* colonnes et chaque entrée $\Pi_{x,y}$ est remplacée par une matrice de permutation *Z*Z* qui peut être ou bien la matrice nulle ou bien une rotation de la matrice d'identité. Une rotation de la matrice d'identité *3*3* par *2* est illustrée dans la Figure 2.2.b. Puisque que chaque sous matrice est soit une matrice nulle, soit la permutation de la matrice d'identité, la matrice *H* peut être représentée sous une forme compacte en plaçant les valeurs de la rotation dans la matrice non nulle et en plaçant des *-1* dans la sous matrice nulle, comme indiqué dans la Figure2. 3.

Cette forme compacte de la matrice *H* est appelée la matrice $H_{Base.}$. La Figure2. 3 illustre la matrice H_{Base} pour le standard WiMAX avec :

$$H = \begin{bmatrix} \Pi_{0,0} & \Pi_{0,1} & \ldots\ldots & \Pi_{0,Y} \\ \Pi_{1,0} & \Pi_{1,1} & \ldots\ldots & \Pi_{1,Y} \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \Pi_{X,0} & \Pi_{X,1} & \ldots\ldots & \Pi_{X,Y} \end{bmatrix} \qquad I = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

Division d la matrice H en différentes sous matrices

Rotation de la matrice d'identité

Figure 2.2 *Représentation de la matrice H*

X = nombre des lignes = *12*

Y = nombre des colonnes = *24*

*Y * Z* =La taille du mot de code = *576*

r = (Y - X)/ Y = Taux de code = ½

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
1	-1	94	73	-1	-1	-1	-1	-1	55	83	-1	-1	7	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
2	-1	27	-1	-1	-1	22	79	9	-1	-1	-1	12	-1	0	0	-1	-1	-1	-1	-1	-1	-1	-1	-1
3	-1	-1	-1	24	22	81	-1	33	-1	-1	-1	0	-1	-1	0	0	-1	-1	-1	-1	-1	-1	-1	-1
4	61	-1	47	-1	-1	-1	-1	-1	65	25	-1	-1	-1	-1	-1	0	0	-1	-1	-1	-1	-1	-1	-1
5	-1	-1	39	-1	-1	-1	84	-1	-1	41	72	-1	-1	-1	-1	-1	0	0	-1	-1	-1	-1	-1	-1
6	-1	-1	-1	-1	46	40	-1	82	-1	-1	-1	79	0	-1	-1	-1	-1	0	0	-1	-1	-1	-1	-1
7	-1	-1	95	53	-1	-1	-1	-1	-1	14	18	-1	-1	-1	-1	-1	-1	-1	0	0	-1	-1	-1	-1
8	-1	11	73	-1	-1	-1	2	-1	-1	47	-1	-1	-1	-1	-1	-1	-1	-1	-1	0	0	-1	-1	-1
9	12	-1	-1	-1	83	24	-1	43	-1	-1	-1	51	-1	-1	-1	-1	-1	-1	-1	-1	0	0	-1	-1
10	-1	-1	-1	-1	-1	94	-1	59	-1	-1	70	72	-1	-1	-1	-1	-1	-1	-1	-1	-1	0	0	-1
11	-1	-1	7	65	-1	-1	-1	-1	39	49	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0	0
12	43	-1	-1	-1	-1	66	-1	41	-1	-1	-1	26	7	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0

Figure2. 3 *La matrice H_{Base} pour le standard WiMax avec un mot de code = 576 , Z = 24 et r = 1/2*

Même s'il a été prouvé dans [MAN03] que la performance des codes structurés est très proche de celle des codes aléatoires, le fait d'ajouter des contraintes pour construire les codes structurés peut dégrader la performance du décodage de ces codes. Par conséquent, il faut bien définir les contraintes au cours du développement des codes structurés afin de maintenir de bonne capacité de correction d'erreurs des codes LDPC. De plus, les codes structurés ne supportent qu'une seule classe des codes LDPC.

Afin de gérer les classes existantes mais également les futures classes des codes LDPC, comme par exemple les codes LDPC non binaire, il est nécessaire de disposer d'approche générique pouvant nativement s'adapter à tout type de code.

B. Résolution des conflits à l'exécution

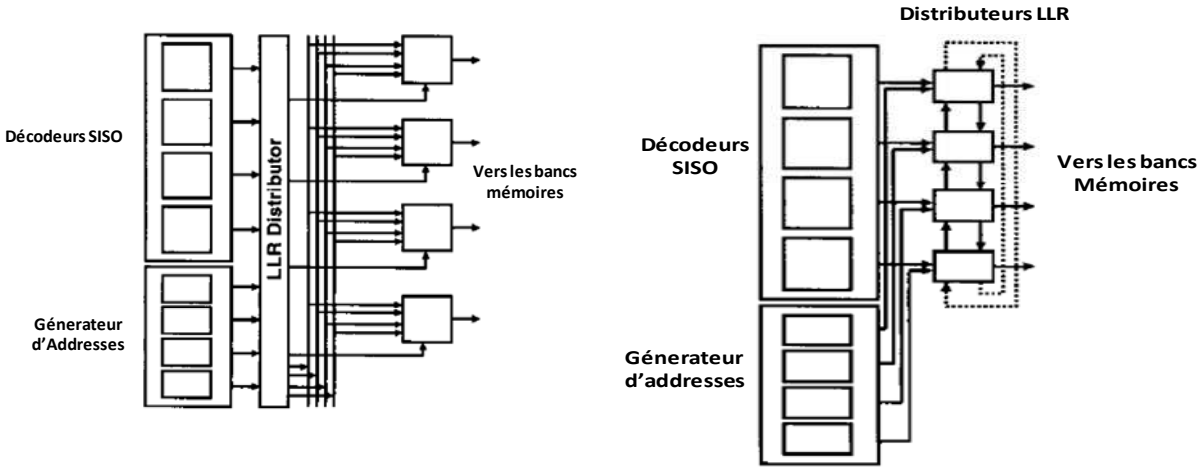
Dans une seconde famille d'approche, le principe consiste cette fois à corriger les éventuels conflits au cours de l'exécution du système. En effet il s'agit par exemple d'assigner les données dans les mémoires de sorte qu'il n'y ait pas de conflits au cours de l'accès dans l'ordre naturel, tandis que dans l'ordre entrelacé, les données conflictuelles seront mémorisées dans le réseau d'interconnexion jusqu'à ce que le bloc ciblé puisse les traiter. Ainsi cette approche peut être appliquée aussi bien dans le cas des Turbo-codes que pour les codes LDPC.

a) Cas des Turbo-codes

L'approche utilisée pour résoudre les problèmes de conflits mémoires des Turbo-codes s'effectue en deux étapes : en premier lieu et quelque soit l'entrelaceur, le réseau d'interconnexion est généré de manière optimisée permettant de réduire les conflits mémoires. Ensuite, des éléments de mémorisation externes sont introduits afin de stocker les données conflictuelles.

Dans [THU02], le réseau d'interconnexion nommé distributeur LLR (Log Likelihood Ratio, rapport logarithmique de vraisemblance) est connecté d'un coté à tous les processeurs et de l'autre à tous les bancs mémoires. Ce distributeur LLR reçoit toutes les données venant des processeurs et détermine leurs bancs mémoire cibles ainsi que leurs adresses dans ces bancs mémoire.

Un entrelaceur appelé « *Tree Interleaver Bottleneck Breaker (TIBB)* » (cf Figure2. 4.a) a été proposé. Ici le distributeur LLR est utilisé comme un routeur et les buffers associés aux mémoires cibles sont utilisés pour la mémorisation des données. Chaque buffer doit avoir accès à toutes les données venant des processeurs afin de déterminer quelle donnée doit être stockée dans sa mémoire locale. Chaque buffer possède *P* connections avec le distributeur LLR afin de pouvoir stocker plusieurs données en un seul cycle. Par conséquent, la complexité du réseau d'interconnexion augmente exponentiellement avec *P*.



Architecture TIBB Architecture RIBB

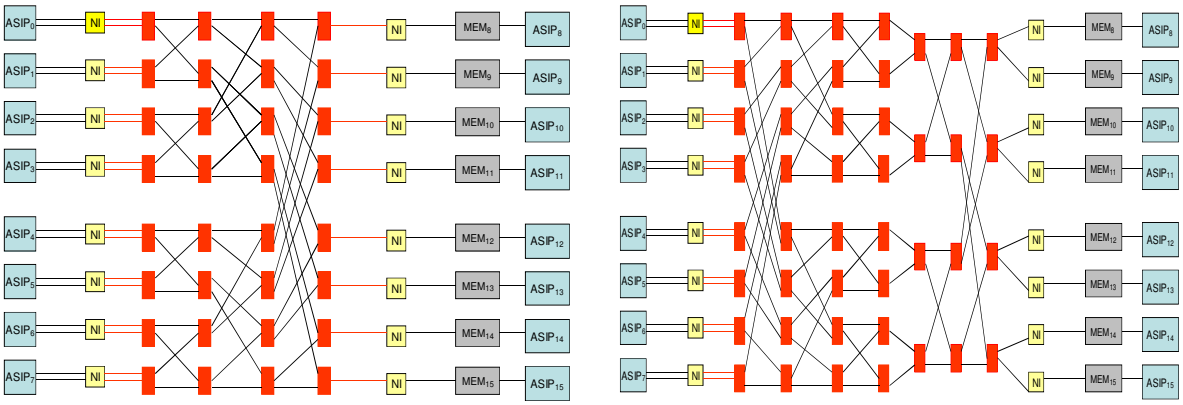
Figure2. 4 *Architecture du distributeur LLR*

Dans [THU02a], l'auteur propose une nouvelle structure appelée « Ring Interleaver Bottleneck Breaker (RIBB)» (voir Figure2. 4.b). Dans cette approche optimisée, chaque processeur a son propre distributeur LLR local, ce qui entraine une architecture moins complexe comparée à celle avec un unique distributeur LLR de la structure TIBB. Ce distributeur doit uniquement décider si la donnée sera stockée dans sa mémoire locale ou bien envoyée vers le distributeur qui est à sa droite ou bien celui qui est à sa gauche. La direction d'une donnée non locale est déterminée en se basant sur la distance la plus courte vers le banc mémoire cible. Comme plusieurs données peuvent cibler le même banc mémoire, le buffer doit être capable de mémoriser plusieurs données en un seul cycle. Comparée à la structure TIBB, l'entrelaceur RIBB réduit la complexité du réseau d'interconnexion et du contrôleur, il introduit toutefois une latence supplémentaire pour transférer les données à la RAM cible.

Afin d'améliorer la latence, une nouvelle structure appelée « General Interleaver Bottleneck Breaker» (GIBB) a été introduite dans le document [THU03]. Dans la structure RIBB, chaque distributeur local LLR est connecté à ses deux voisins distributeurs LLR, tandis que dans la structure GIBB, chaque distributeur LLR peut être connecté à n'importe quel nombre de distributeurs. Cette approche augmente la capacité du réseau et par conséquence le débit du décodeur. La topologie de GIBB est représentée comme un graphe direct avec ses informations de routage qui utilisent le plus court chemin pour transférer les données entre les différents nœuds ou les distributeurs LLR. Toutefois, le calcul du chemin le plus court dans un graphe en général est un problème NP-complet et il n'y a pas d'algorithme optimal permettant de trouver le chemin de routage le plus court. En outre, si le degré du parallélisme augmente l'architecture matérielle du distributeur LLR est encore plus complexe, et les buffers sont plus grands. Tout ceci rend ces approches coûteuses en termes de surface et de latence.

Afin de réduire l'interconnexion et la taille des buffers, un réseau sur puce à commutation par paquet a été proposé pour résoudre les conflits en cours d'exécution. Dans [NEE05], l'auteur propose un réseau cubique, ainsi qu'un réseau de type Mesh et un réseau thorique. Les paquets introduits dans ces réseaux contiennent des informations sur le processeur cible, la mémoire cible et les données. Toutefois, ces topologies sont caractérisées par une flexibilité limitée, et ne sont pas parfaitement adaptés au réseau de communication sur puce à haut débit. En outre la complexité des routeurs augmente avec l'augmentation du parallélisme, ceci est dû à la complexité de l'architecture de gestion du buffer pour mémoriser les données conflictuelles.

Afin de répondre aux contraintes de débits et améliorer la flexibilité dans les réseaux de communication sur puce flexibles, deux réseaux multi étages hétérogènes ont été étudiés dans [MOU07]. Le réseau papillon qui est un réseau d'interconnexion sur puce multi étages avec des liaisons unidirectionnelles possède deux routeurs en entrée ainsi que deux routeurs en sortie (Figure 2**.5**.a).



Réseau Papillon

Réseau Benes 2N-N

Figure 2. 5 *Réseau multi étage hétérogène*

Comparée aux réseaux que nous venons de mentionner, le réseau papillon a deux avantages: en premier lieu, le réseau est évolutif car un réseau de diamètre *D* peut être construit à partir d'un réseau de diamètre *D-1*. En second lieu, l'algorithme de routage de paquets est simplifié par l'utilisation des bits d'adresses de destination pour sélectionner le port de sortie du routeur à chaque étage du réseau.

Ce réseau est composé par des routeurs et des interfaces réseaux: les routeurs stockent les paquets conflictuels dans des FIFOs et l'interface réseau inclut les informations d'entrelacement continues dans l'entête du paquet et venant de l'unité de traitement. Cependant, le réseau papillon fournit un unique chemin entre chaque source et chaque destination, il en résulte un manque de diversité au niveau des chemins. Ceci a pour conséquence l'augmentation de la complexité des architectures de gestion des paquets conflictuels, et donc la surface du silicium et le coût du réseau.

Un autre réseau étudié dans [MOU07] est le réseau de Benes qui consiste en la concaténation de deux réseaux de papillons de directions opposées. Le réseau de Benes offre une plus grande diversité de chemins ainsi que toute les permutations possibles entre les entrées et les sorties. Ceci peut être nécessaire pour la construction d'un réseau flexible, capable de réaliser tout type d'entrelaceurs dans les turbo-décodeurs. Cependant, ce réseau permet d'éviter les conflits entre les paquets si et seulement si tous les paquets ont des destinations différentes, ce qui n'est potentiellement pas le cas dans les turbo-décodeurs. Afin d'optimiser le réseau de Benes pour qu'il soit utilisé dans les turbo-décodeurs, l'auteur propose une nouvelle topologie ainsi qu'un algorithme de routage (voir Figure 2. **5**.b). Dans cette topologie le nombre de routeurs dans le premier étage est *2N* tandis que le nombre de routeurs dans le deuxième étage est réduit à *N*. Les algorithmes de routage sont basés sur le fait que les paquets destinés aux différents routeurs sont transmis au même cycle. Par conséquent, des prétraitements s'avèrent nécessaires pour pouvoir ordonnancer tous les paquets en attribuant à chacun d'eux un intervalle de temps. Le format du paquet est le même que celui du réseau papillon et la mémoire FIFO est remplacée par des registres dans le routeur. Des compteurs sont introduits dans l'interface réseau dont le rôle est d'ordonnancer la transmission des paquets.

L'un des inconvénients majeurs du réseau papillon est la complexité des architectures de gestion des buffers, qui engendre une augmentation significative de sa surface. Au contraire, un réseau de

Benes nécessite un pré calcul des chemins de routage ainsi qu'un ordonnancement des paquets, ceci n'est pas considéré comme une solution convenable pour la réalisation d'architectures multistandards ou des architectures des décodeurs flexibles. Par ailleurs, la diversité des chemins proposée par ces réseaux n'est pas suffisante pour gérer les conflits des paquets dans certains cas d'applications.

Afin de faire face à toutes ces limitations, un nouveau réseau a été présenté [MOU08] appelé réseau d'interconnexion de Bruijn binaire. Ce réseau est évolutif et permet de réaliser toute permutation possible. Pour faciliter la présentation de ce réseau, l'auteur introduit dans le document [BRU46] un graphe de Bruijn binaire. Un graphe de 16 nœuds est présenté dans la Figure2. 6 pour décrire les différents chemins entre les nœuds.

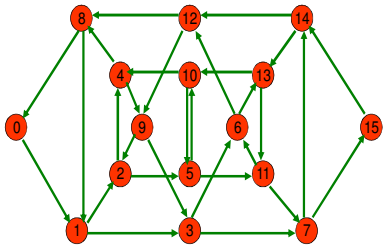


Figure2. 6 *Graphe de Bruijn binaire avec 16 nœuds*

Grâce à la diversité des chemins offerts par ce réseau, en cas de conflits de communication les paquets conflictuels changent de chemin jusqu'à atteindre le processeur ou bien la mémoire cible, ceci permet de s'affranchir des techniques de temporisation, de mémorisation de ces paquets dans des buffers. L'inconvénient de ce réseau est la grande complexité de l'architecture de contrôle nécessaire à la gestion des éléments conflictuels, qui engendre un surcoût matériel. En outre, le retard introduit par le mécanisme de gestion des conflits dégrade les performances en termes de débit.

Récemment, de nouveaux réseaux d'interconnexion optimisés et à faible coût ont été proposés pour implémenter des entrelaceurs utilisés dans les derniers standards de télécommunication. Dans [WON10], l'auteur propose dans le système 3GPP LTE un réseau d'interconnexion multi étages basé sur un barrel shifter. Grâce aux caractéristiques spécifiques des permutations réalisées par l'entrelaceur QPP, la connexion entre chaque banc mémoire et ses décodeurs SISO est établit en décalant chaque sous blocs un certain nombre de fois. Ce nombre peut être variable mais l'important est qu'il soit le même pour tous les sous blocs. Ainsi, puisque le décalage sera le même, il est possible d'utiliser un barrel shifter comme composant de permutation. Ce réseau d'interconnexion est caractérisé par son court chemin ainsi que son mécanisme simple de control de routage.

L'atout majeur de ce réseau est qu'il est capable de réduire le coût matériel du décodeur pour les applications à haut débit et à faible consommation. Cependant, le réseau proposé ne peut être implémenté que dans le cas d'un entrelaceur QPP dont le nombre de SISO est une puissance de deux, en outre l'approche ne peut pas être appliquée sur n'importe quelle règle d'entrelacement.

b) Cas des codes LDPC

Les concepteurs des décodeurs LDPC accordent une grande attention à la problématique du transfert de l'information entre les nœuds de variables et les nœuds de parité. Ainsi, différents réseaux d'interconnexion évolutifs et flexibles ont été proposés dans la littérature pour résoudre les problèmes de conflits mémoires à l'exécution. Néanmoins avoir des réseaux flexibles et évolutifs dans les codes LDPC a pour conséquence de dégrader la latence et le coût matériel du système.

Dans [THE05], l'auteur propose une approche basée sur un réseau sur puce, les nœuds de variables et les nœuds de parité jouent le rôle des processeurs. L'approche utilise un réseau sur puce (NoC) avec

une topologie 2D mesh pour les faire communiquer entre eux. En se basant sur la matrice *H* du code LDPC, l'approche génère la configuration des données qui contient l'information sur le nombre des nœuds (VN et CN) qui sont alloués à un processeur particulier ainsi que l'information sur la connexion entre les différents processeurs. Chaque processeur a une mémoire dédiée pour stocker la configuration des données. Le paquet transmis par le processeur contient l'information sur la donnée ainsi que l'adresse de l'émetteur et celle du récepteur qui sont utiles pour le routage de la donnée sur le réseau. Afin de minimiser le coût introduit par le routage l'auteur propose un algorithme d'assignation intelligent pour transposer le graphe de Tanner sur le réseau physique.

Dans [KIE03], les concepteurs utilisent un réseau hétérogène appelé réseau de distribution de l'information « Message Distribution Network (MDN) », sa topologie est basée sur un graphe généré de façon aléatoire et il est utilisé pour la résolution des conflits mémoire à l'exécution. L'échange entre les nœuds de variable et les nœuds de parité est effectué en rajoutant l'adresse de destination à l'entête du message généré. Plusieurs mémoires sont utilisées pour stocker les données, les adresses, le message produit ainsi que les valeurs reçues par le canal. Le mécanisme de gestion de conflits dégrade le débit ainsi que la latence. Par ailleurs l'implémentation du décodeur LDPC avec le réseau MDN nécessite un surcoût important en terme de surface et de consommation de tout le système. Cependant, ces réseaux sur puce favorisent quelque peu la conception des architectures flexibles et évolutives des décodeurs LDPC.

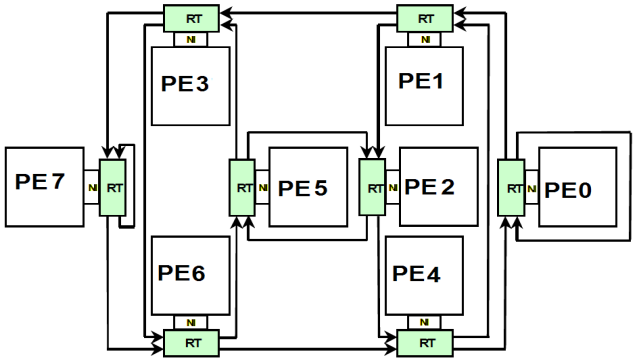


Figure2. 7 *Réseau de Bruijn avec 8 processeurs, des routeurs et des interfaces réseau*

Dans [MOU08] l'auteur propose d'utiliser le réseau de Bruijn binaire (voir Figure2. 7) permettant d'avoir un décodeur évolutif avec une latence et une surface raisonnable. Le réseau de Bruijn offre une grande flexibilité en termes d'échange d'information entre les nœuds de variables et les nœuds de parité. L'attribution des nœuds de variables et des nœuds de parité à chaque processeur dépend du rendement du code, des degrés des nœuds et de la taille des mémoires extrinsèques. En outre, puisque la somme des degrés des nœuds de variable est égale à la somme des degrés des nœuds de parité, la moitié des processeurs sont attribués aux nœuds de parité et le reste est attribué aux nœuds de variable. Afin d'implémenter les arcs du graphe de Tanner, l'entête de chaque paquet contient l'information sur le processeur cible ainsi que l'adresse d'écriture dans la mémoire de destination. Un algorithme de calcul de plus court chemin est appliqué pour calculer le routage des paquets. Si deux paquets sont destinés vers la même sortie, l'un des deux paquets sera dévié au lieu de le bloquer, ou bien de le mémoriser dans un buffer, ceci aura pour conséquence de réduire la taille du routeur.

Pour conclure, cette famille d'approche est basée sur l'implémentation des réseaux d'interconnexion flexibles. En cas de conflits mémoire, les données conflictuelles sont momentanément stockées dans des éléments de mémorisation ou bien déviées au sein du réseau jusqu'à ce que le bloc cible puisse les traiter. L'avantage de ce type d'approche réside dans sa généricité puisque la majorité des réseaux implémentés sont capables de supporter n'importe quel

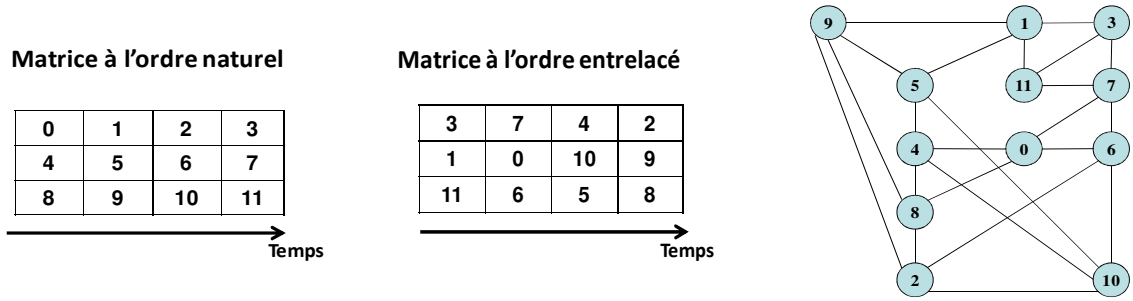
standard de communication. Néanmoins le mécanisme de gestion de conflits introduit a pour conséquence de dégrader la latence et le coût matériel du système.

C. Résolution des conflits à la conception

Un troisième type d'approche consiste à effectuer un pré traitement pour déterminer le placement mémoire pour chaque donnée afin de réaliser des accès parallèles sans conflit des processeurs vers les bancs mémoires. L'atout majeur de ces approches réside dans le fait que si le réseau d'interconnexion supporte toutes les permutations, alors elles trouveront un placement des données en mémoires sans conflit. Cependant, cette approche nécessite un prétraitement afin de placer les données dans différents bancs mémoires pour toutes les tailles de blocs et tous les degrés de parallélisme requis par le système supportant un des standards.

a) Graphes de conflits et Turbo-Codes

Afin d'illustrer l'utilisation des graphes de conflits d'allocation mémoire dans le cas des Turbo-Codes la Figure 2. 8.a modélise les accès aux données d'un entrelaceur (ordre naturel et ordre entrelacé). Le graphe de conflit correspondant est représenté en Figure 2. 8.b. Les données d'une même colonne dans la matrice de l'ordre naturel, et respectivement de l'ordre entrelacé, doivent être accédées en parallèle, elles doivent donc être adjacentes dans le graphe de conflit. Par exemple, les données *0, 4* et *8* doivent être accédées à l'instant t_1 (ordre naturel), elles sont donc connectées par des arcs dans le graphe de conflit.



Matrices d'accès aux données — Graphe de conflit

Figure 2. 8 *Graphe de conflit pour les turbo-décodeurs*

Dans une telle situation, un algorithme de coloriage de graphe devrait être capable de trouver un placement des données dans les bancs mémoire, sans conflit et utilisant 3 couleurs (taille de la plus grande clique du graphe). Néanmoins les heuristiques proposées dans la théorie des graphes sont incapables d'optimiser le coût du matériel de l'entrelaceur. La Figure 2. 9 illustre l'architecture de l'entrelaceur générée après avoir trouver une allocation mémoire sans conflit.

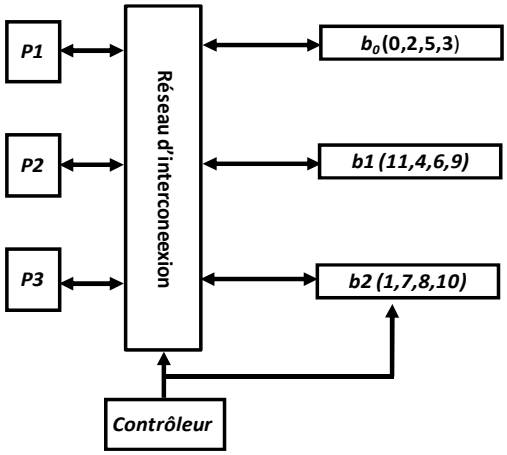
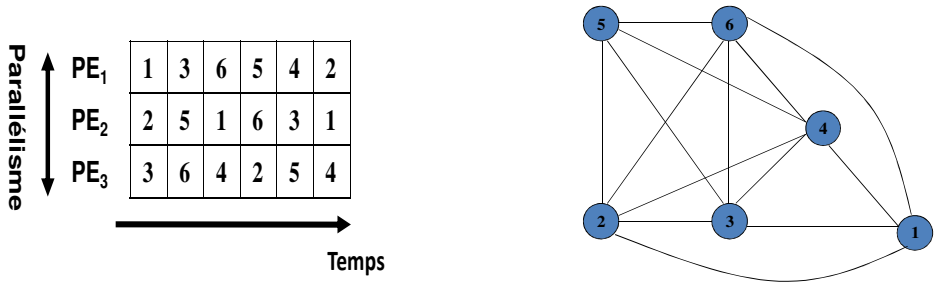


Figure 2. 9 *Architecture résultante*

b) Graphes de conflits et codes LDPC

La Figure 2. 10.b illustre un graphe de conflits généré à partir de l'ordre d'accès présenté dans la Figure 2. 10.a. Il est impossible de trouver un coloriage dont le nombre de couleurs est celui limité par le concepteur, car le graphe résultant *G* montre qu'il faut plus de *P* couleurs ou bancs mémoires, où *P* est le nombre des éléments de traitements dans le système pour colorer tous les nœuds. Il en résulte une architecture dans laquelle plus de *P* bancs mémoires sont utilisés pour réaliser les accès parallèles requis.



(a) Matrice d'accès aux données (b) Graphes de conflits

Figure 2. 10 *Graphe de conflit pour le décodeur LDPC*

Une fois appliqué le coloriage des nœuds au graphe illustré dans Figure 2. 10.b, l'architecture résultante est présentée dans la Figure 2. 11 dans laquelle *5* bancs mémoires sont nécessaires pour stocker 6 données**,** cependant seuls 3 processeurs sont utilisés en parallèles pour traiter les données.

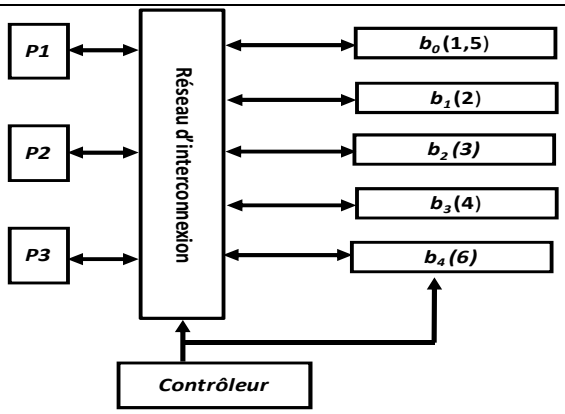


Figure 2. 11 *Architecture résultante*

Afin de trouver un nombre optimal de bancs mémoires, nous exploiterons dans cette thèse le concept appelé « Double Memory Mapping » [CHA10]. L'idée est ici que l'allocation mémoire de chaque donnée est divisée en deux allocations : la première est dite allocation en *lecture* qui représente l'accès en lecture à la donnée, tandis que la seconde est appelée allocation en *écriture* et exprime l'accès en écriture de la donnée. On parlera par la suite de *semi-colonne* de lecture ou de *semi-colonne* d'écriture. Nous verrons que grâce à ce concept le nombre de bancs mémoires nécessaire peut être réduit au nombre de processeurs utilisés en parallèle. Ainsi, on réduit la complexité globale du système. Les approches que nous proposons dans cette thèse sont basées sur ce concept.

c) Approches de placement mémoire pour les Turbo-codes

Sans contrainte de réseau

Un premier algorithme [TAR04], a été proposé pour résoudre les problèmes des conflits mémoire des turbo-codes à la conception. Dans cette méthode l'auteur utilise une méta heuristique dite de « recuit simulé » pour déterminer un placement de données dans des bancs mémoires sans conflits. Les modèles mathématiques et les contraintes d'assignation reliées à ce problème sont détaillés dans [TAR04].

Par exemple dans la Figure2. 12.a, les données *1, 6, 11, 16, 21* doivent être accédées en parallèle au premier cycle de traitement en ordre naturel. La matrice dans la Figure2. 12.b représente les accès en ordre entrelacé ; à chaque cycle de traitement (i.e. à chaque colonne de la matrice) est associé une lettre appelée « *tuile* ». Dans [TAR04], cette tuile est utilisée pour « recouvrir » les cases de la matrice d'accès en ordre naturel (voir Figure2. 12.c).

Ainsi les données reposant sous une même tuile sont accédées en même temps en ordre entrelacé, et sous ces tuiles, puisque que nous avons là la matrice d'accès en ordre naturel, les données d'une même colonne sont accédées en même temps lors de traitement en ordre naturel. Par conséquent, afin d'avoir une allocation mémoire valide et donc sans conflits, tous les bancs mémoire assignés aux données d'une même tuile de la matrice tuilée et dans chaque colonne de cette matrice doivent être différents.

Matrice de l'ordre naturel

P1	1	2	3	4	5
P2	6	7	8	9	10
P3	11	12	13	14	15
P4	16	17	18	19	20
P5	21	22	23	24	25

Matrice de l'ordre entrelacé

P1	25	10	22	3	1
P2	14	11	5	6	2
P3	15	13	7	16	20
P4	24	18	8	19	17
P5	12	21	4	23	9
Tile	A	B	C	D	E

Matrice tuilée

E	E	D	C	C
D	C	C	E	B
B	A	B	A	A
D	E	B	D	E
B	C	D	A	A

Figure2. 12 *Principe de Tuilage*

L'algorithme présenté dans [TAR04] est divisé en deux étapes :

Première étape : Il s'agit de la construction d'une matrice d'affectation préliminaire avec les propriétés suivantes : « Chaque colonne ainsi que chaque tuile contient au maximum un élément égal à chaque symbole dans *{1,….,P}* ». La construction de cette matrice préliminaire est encore améliorée par une initialisation gloutonne de la matrice d'affectation [TAR05] de la manière suivante : « Pour *i=1,….P,* lire la $i^{ème}$ ligne de la matrice d'affectation de gauche à droite et affecter la valeur de l'élément courant à *i* de telle sorte qu'il y aura pas de collision avec les autres éléments de la même ligne, sinon garder la case vide ».

Deuxième étape : Dans cette étape, l'approche utilise un algorithme de recuit simulé sur la matrice d'affectation préliminaire afin de remplir toutes les cases vides. Pour ceci, l'algorithme injecte une collision soit dans une colonne ou dans une tuile, et résout cette collision à chaque étape en introduisant probablement une autre collision. L'idée principale est que l'ensemble va se stabiliser et l'on pourra ensuite continuer avec les conflits restants.

Le problème de cet algorithme se situe dans la seconde étape. Il n'est pas possible de déterminer le temps nécessaire pour avoir une matrice d'allocation mémoire complète sans conflit. De fait on ne sait pas prédire la convergence de l'algorithme, même si les auteurs disent avoir prouvé cette convergence. De plus, cette approche ne supporte aucune contrainte sur l'architecture de réseau d'interconnexion cible et ne sait pas optimiser le coût de l'architecture de contrôle.

Dans [LIN10], l'auteur présente une réallocation optimisée des adresses mémoires (OPMM) dans laquelle des règles d'échanges sans collision sont définies afin de compléter la procédure du recuit simulée plus rapidement que celle utilisée dans la méthode traditionnelle présentée dans le document [TAR04]. L'OPMM accélère la procédure du recuit simulé de deux façons : d'abord l'algorithme sélectionne des paires de données qui ont besoin d'échanger leurs bancs mémoires afin d'effectuer plusieurs étapes d'OPMM en une seule itération, ensuite durant les futures itérations, les paires de données sélectionnées échangent leurs informations de bancs de telle sorte que les données peuvent seulement varier entre deux bancs mémoires au lieu de *P* bancs mémoire.

Les expériences réalisées montrent que la procédure OPMM est capable de trouver une allocation mémoire sans conflit plus rapidement. Cependant, la méthode est basée sur une métaheuristique et la complexité ainsi que le temps d'exécution de l'algorithme sont inconnus. De plus, comme [TAR04], il ne gère pas la contrainte sur le réseau d'interconnexion, ni l'optimisation du contrôleur.

Avec contrainte de réseau

Dans [CHA10a], les auteurs présentent une nouvelle approche simplifiée appelée « Static Address Generation Easing» (SAGE), cette méthode rajoute des contraintes supplémentaires permettant d'orienter l'allocation mémoire afin de cibler une architecture d'entrelaceur particulière. Dans cette approche, deux matrices vides appelées « SAGE Mapping Matrices » sont utilisées pour stocker les informations des bancs mémoire durant l'exécution de l'algorithme. Ces deux matrices (MAP_{Nat}, MAP_{Int}) sont de même ordres et elles sont illustrées dans la Figure2. 13.

Afin de cibler une architecture particulière sans conflit mémoire, deux contraintes sont définies et doivent être respectées durant l'exécution de l'algorithme d'allocation mémoire. Premièrement, chaque colonne des matrices d'allocation doit contenir des bancs mémoire différents. Deuxièmement, chaque colonne respect e des règles de pilotage du réseau d'interconnexion ciblé si la loi de permutation le permet.

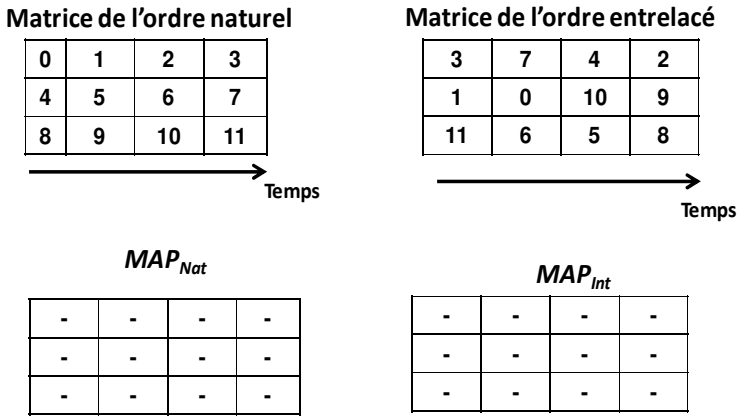


Figure2. 13 *Matrices utilisées dans SAGE*

L'algorithme commence par effectuer une première assignation à la première colonne de MAP_{Nat}, puis l'algorithme met à jour dans MAP_{Int} les données qui viennent d'être assignées dans MAP_{Nat}. Ensuite, à chaque itération l'algorithme sélectionne la colonne la plus contrainte dans l'une des deux matrices (la colonne ayant le plus des données déjà assignées), puis remplit la colonne avec des informations d'assignations respectant les contraintes définies et met à jour ces informations dans la seconde matrice jusqu'à ce que les colonnes des deux matrices soient complètement remplies.

Cette approche, si elle se révèle capable de trouver un placement des données en mémoire sans conflit doit faire face à plusieurs limitations : en premier lieu, si la règle d'entrelacement ne permet pas nativement de respecter l'architecture cible définie pour le réseau d'interconnexion, alors [CHA10a] ne sera pas capable de générer l'architecture voulues par le concepteur. De plus, cette approche n'offre pas de solution pour l'optimisation de l'unité de contrôle du système.

Dans [SAN11a], le problème est formulé au travers de deux matrices : une matrice qui exprime l'accès à l'ordre naturel et l'autre qui exprime l'accès à l'ordre entrelacé (voir Figure2. 14). Chaque matrice a *P* lignes qui montrent les éléments de traitements et *T/2* colonnes qui montrent les instants d'accès. Le problème d'allocation mémoire est modélisé par un graphe biparti. Ensuite le graphe est transformé en un problème de transport auxquels divers algorithmes de l'état de l'art peuvent être appliqués pour trouver l'allocation mémoire des données.

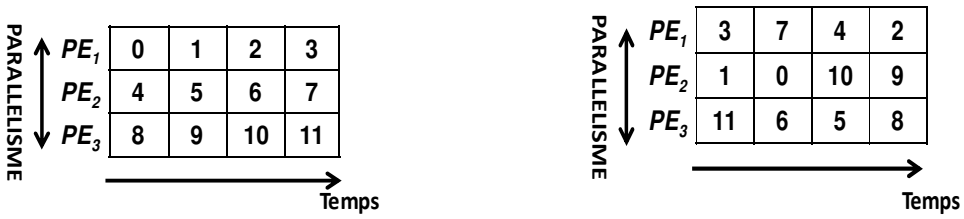


Figure2. 14 *Matrice d'accès aux données pour les turbo-codes*

Le graphe biparti $G = (T \cup D, E)$ est illustré dans la Figure2. 15, avec T l'ensemble des sommets qui représentent tous les instants d'accès aux données et D est l'ensemble des sommets qui représentent toutes les données utilisées dans le calcul. Un arc (t,d) relie la donnée d à l'instant t si d doit être traité à l'instant t.

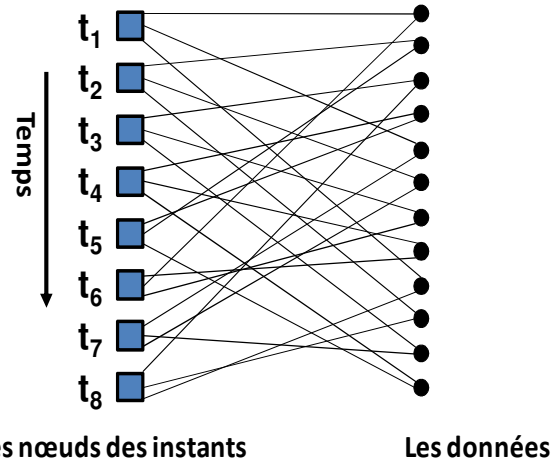


Figure2. 15 *Représentation du graphe biparti pour la Figure2. 14*

Le problème d'allocation mémoire est ensuite transformé en un problème de « transport ». Une matrice est construite en respectant un réseau d'interconnexion particulier. Une fois la matrice construite, une approche basée sur la programmation linéaire est utilisée pour trouver une allocation de mémoire sans conflit.

Cette approche permet de trouver une assignation mémoire sans conflits et de concevoir un réseau d'interconnexion simple à condition la règle d'entrelacement le permette. Ainsi, cette approche doit faire face aux mêmes limitations que [CHA10a].

d) Approches de placement mémoire pour les codes LDPC

Les approches présentées dans les documents [TAR04] et [LIN10] peuvent également être utilisées dans le cas des codes LDPC, à condition (cf. [TAR04]) de gérer les données en mode « *Single Static Assignment* » (SSA), c'est-à-dire que chaque accès à une donnée sera considéré comme indépendant des autres et devra être mémorisé dans une case mémoire spécifique. Autrement dit, si une donnée est accédée 4 fois, plutôt que de faire 4 accès dans une même case mémoire, il faudra utiliser 4 cases mémoires distinctes. Par conséquent, l'ajout des bancs mémoires supplémentaires devient indispensable afin de trouver une allocation mémoire sans conflit dans certains cas d'application.

L'implémentation de l'algorithme d'allocation mémoire présenté dans le document [TAR04] est détaillée dans le document [QUA06]. L'architecture se compose de deux cross bars ou bien deux réseaux de Benes afin de pouvoir gérer l'échange des données entre les différents processeurs. Un cross bar est utilisé pour transmettre les informations aux nœuds de variable et un autre est utilisé pour transmettre les informations aux nœuds de parité. L'algorithme «Traditional Belief Propagation » a été modifié pour avoir l'algorithme « Low-Traffic Belief Propagation » (LTBPA) dans le but d'utiliser un seul réseau d'interconnexion, néanmoins le LTBPA n'est pas l'algorithme standard utilisé dans les implémentations des décodeurs actuels.

Ces approches font quoi qu'il arrive face à un surcoût mémoire important du fait de leur mode de fonctionnement en SSA. De plus, elles ne peuvent pas être utilisées pour cibler un réseau d'interconnexion particulier, ni optimiser le coût du contrôleur.

Dans [CHA10], les auteurs présentent une technique basée sur l'utilisation de bancs mémoire potentiellement distincts pour la lecture d'une donnée à un cycle de traitement donné et l'écriture du résultat correspondant. Ainsi selon cette approche, chaque donnée d_i est assigné à deux bancs mémoires, un à partir desquels le processeur lit la donnée et un autre dans le quel le processeur écrit le résultat. C'est pourquoi dans l'exemple de la Figure2. 16 deux cases sont associées à chaque donnée d_i.

Afin d'obtenir une allocation mémoire valide, si une donnée est accédée plusieurs fois, son $i^{ème}$ accès en lecture doit être assigné au même banc mémoire que celui assigné à son $(i-1)^{ème}$ accès en écriture. Deux autres contraintes doivent également être vérifiées : dans chaque colonne tous les bancs mémoires doivent être différents (accès mémoire sans conflit) et le premier accès en lecture d'une donnée d_i doit être assigné au même banc mémoire que celui assigné à son dernier accès en écriture

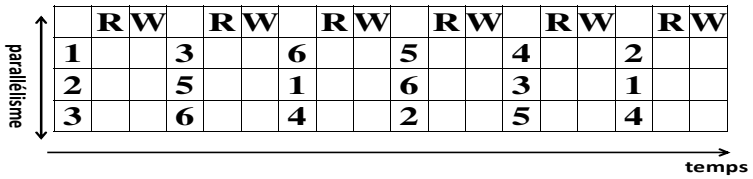


Figure2. 16 *Matrice d'assignation de l'approche basée sur la lecture multiple et l'écriture multiple*

L'algorithme commence par effectuer une première assignation des bancs mémoires aux données situées dans les colonnes de lecture et d'écriture du cycle de traitement t_1. Ensuite, l'algorithme continu l'assignation de l'accès en lecture et en écriture des données situées dans les colonnes qui suivent, tout en respectant les contraintes imposées. Ceci se poursuit tant qu'aucun conflit n'apparait. En cas de conflit l'algorithme effectue un retour en arrière pour aller à l'occurrence la plus poche de la donnée conflictuelle afin de changer son assignation et résoudre ainsi le conflit. Ce processus continue jusqu'à l'obtention d'une assignation mémoire valide de toutes les données.

Cet algorithme décrit une nouvelle approche pour résoudre des problèmes difficiles présents dans le cadre des codes LDPC. Cependant, cet algorithme est basé sur une approche récursive et le temps d'exécution est inconnu. De plus il ne sait pas cibler un réseau d'interconnexion particulier, ni optimiser le coût du contrôleur.

Dans le document [SAN10], l'auteur présente un algorithme permettant de résoudre le problème de conflits mémoire pour des codes LDPC en se basant sur les graphes bipartis. La formulation du problème est basée sur le concept « Double Memory Mapping » [CHA10]. Puis l'accès aux données est modélisé par un graphe biparti (cf. Figure2. 17). Ensuite un algorithme d'allocation de mémoire est appliqué pour colorer les arcs de ce graphe tout en respectant les contraintes. Afin de faciliter le coloriage des arcs, le graphe biparti est divisé en sous graphes et chaque sous graphe est traité de manière séparée.

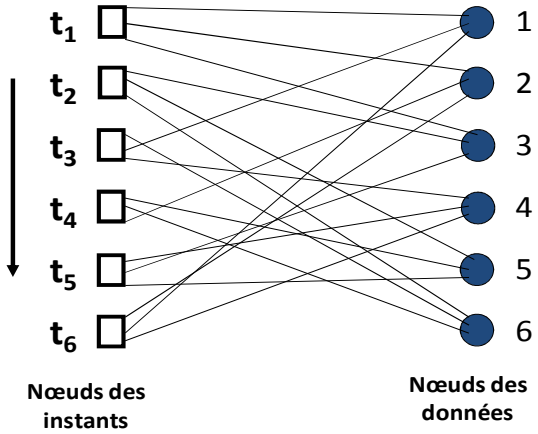
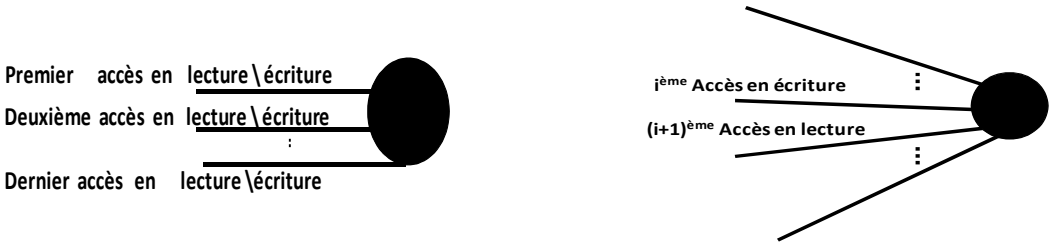


Figure2. 17 *Modélisation biparti du problème d'allocation de la mémoire pour les codes LDPC*

Afin de diviser le graphe en sous graphes un algorithme de partitionnement est appliqué. Dans cet algorithme certaines contraintes de partitionnements sont imposées. L'algorithme est basée sur deux processus : un processus de parcours qui permet de construire un chemin dans le graphe de manière aléatoire et un autre processus d'élimination dont le rôle est d'éliminer tous les arcs qui contredisent les contraintes de partitionnement. Une fois le partitionnement effectué, un algorithme de coloriage des arcs du graphe biparti est appliqué pour chaque partition.



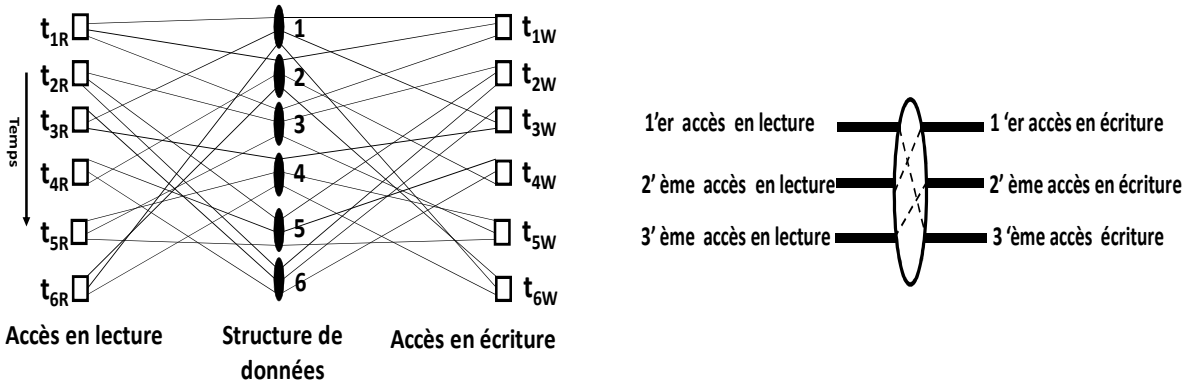
Les arcs d'une donnée **(b)** Propriété de placement

Figure2. 18 *Représentation des arcs d'un nœud d'une donnée du graphe biparti*

Grâce à la modélisation du problème d'allocation mémoire par un graphe biparti, et à l'utilisation du concept de « allocation mémoire double », cette approche est capable de concevoir un entrelaceur sans conflits mémoire avec un nombre réduit de bancs mémoire. Néanmoins, elle ne cible pas un réseau d'interconnexion particulier, et n'est pas non plus capable d'optimiser le coût du contrôleur.

Deux autres approches [SAN11] [SAN11a] ont été également proposées par les mêmes auteurs. Elles se basent sur l'utilisation d'un graphe triparti. L'avantage de ses méthodes réside dans le fait qu'un graphe triparti peut être converti en un graphe biparti sur lequel n'importe quel algorithme de coloriage d'arc peut être appliqué. Il est alors possible d'exploiter des algorithmes de coloriage d'arcs efficaces.

La première approche [SAN11] est dédiée à la résolution les problèmes d'allocations mémoires des codes LDPC. La modélisation du graphe tripartie est basée sur la matrice d'accès aux données illustrée dans Figure2. 14. La Figure2. 19 illustre un graphe triparti. Chaque graphe construit est partitionné en différents sous graphes colorés de manière indépendante.



a. b.

Figure2. 19 *Graphe tripartie pour d'accès aux données illustrées dans Figure2. 14*

Dans la seconde approche [SAN11a], les deux techniques d'allocation mémoire simple et double sont modélisées par un graphe triparti. Ensuite, les arcs corrélés (c'est-à-dire devant être assignés au même banc mémoire, cf Figure2. 19) dans ce graphe sont reliés et les nœuds de données sont supprimés afin de générer un graphe biparti. Ainsi, n'importe quel algorithme de coloriage des arcs d'un graphe biparti peut être appliqué. Une comparaison de la complexité de ces différentes approches, montre que [SAN11a] est la meilleure en termes de généricité et également de complexité puisqu'elle peut être exécutée en un temps polynomial. Cette approche est la plus rapide de toute mais elle ne sait pas cibler un réseau ni réduire la complexité du contrôleur.

III. Bilan

Dans ce chapitre nous avons présenté différentes techniques présentes dans la littérature permettant de résoudre les problèmes de conflits mémoires présents dans les turbo-codes ainsi que les codes LDPC. Une première approche consiste pour le concepteur à développer sa propre loi de permutation tout en prenant en considération les contraintes architecturales. Ceci a pour conséquence de générer un réseau d'interconnexion simplifié mais qui ne peut pas être utilisé pour tout standard de communication. En outre le code développé ne présente pas toujours de bonne capacité de correction d'erreur. Afin de réaliser des décodeurs plus flexibles capables de supporter toute loi de permutation, différentes architectures basées sur des réseaux sur puces (NoC) ont été proposées dans la littérature pour résoudre les problèmes de conflits mémoire au cours de l'exécution. Néanmoins, l'utilisation de cette technique est limitée dans la pratique car ces réseaux sur puce, ainsi que les éléments de mémorisation supplémentaires, imposent un surcoût important et dégradent la latence. Une troisième approche consiste à placer les données dans des bancs mémoires de telle sorte que les éléments de traitements peuvent accéder en parallèle aux données dans les bancs mémoires sans aucun conflit. Cette technique est capable de résoudre le problème d'assignation mémoire pour tout type d'entrelacement. Certaines approches offrent la possibilité aux concepteurs de cibler un réseau

particulier mais sont limitées à certaines applications, de plus elles génèrent des contrôleurs qui sont souvent complexes et imposant un surcoût matériel important.

	Toute loi d'entrelacement	Compatible Turbo-Code	Compatible LDPC	Contrainte de réseau	Optimisation du contrôle
[GNA03]	-	Oui	-	Oui	-
[THU03]	Oui	Oui	Oui	-	-
[MOU07]	Oui	Oui	Oui	Oui	-
[TAR04]	Oui	Oui	Oui	-	-
[LIN10]	Oui	Oui	Oui	-	-
[CHA10a]	Oui	Oui	-	Oui	-
[CHA10]	Oui	Oui	Oui	-	-
[SAN11]	Oui	Oui	Oui	-	-
[SAN11a]	Oui	Oui	-	Oui	-
[SAN10]	Oui	Oui	Oui	-	-
Coloriage	Oui	Oui	-	-	-
Notre contribution	***Oui***	***Oui***	***Oui***	***Oui***	***Oui***

Figure2. 20 *Tableau comparatif des approches de l'état de l'art*

Dans cette thèse, nous proposons des solutions d'allocation mémoires sans conflits valables pour tout standard de communication et respectant la contrainte de réseau définie par le concepteur. Ce qui permet en conséquence de générer des réseaux optimisés selon les désidératas de ce dernier. Ces solutions sont également capables d'optimiser le contrôleur mémoire, tout en gardant de bonnes performances en terme de latence du système. Actuellement, aucune approche de l'état de l'art ne prend en compte tous ces aspects en même temps (cf Figure2. 20).

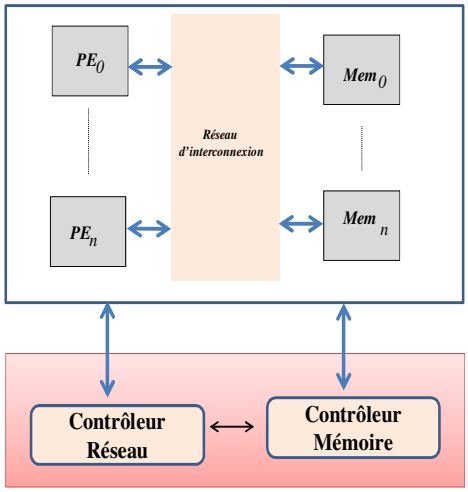


Figure2. 21 *Rappel de l'architecture cible*

En résumé, notre contribution dans le cadre de cette thèse porte sur deux axes principaux qui seront développés dans les chapitres suivant :

- Définir une approche de placement de données en mémoire sans conflit et garantissant au concepteur que l'architecture qu'il souhaite pour le réseau d'interconnexion sera respectée.
- Parvenir à optimiser le coût de l'architecture du contrôleur du système résultant.

-

Chapitre 3
Placement mémoire sous contrainte de réseau

Sommaire

Dans le chapitre précédent, nous avons introduit une étude des solutions existantes dans la littérature permettant de répondre au problème de placement mémoire sans conflit pour des architectures de codeurs/décodeurs parallèles. Certains types d'approches consistent à trouver la bonne assignation des données aux bancs mémoire à la conception, de manière à éliminer tous risques de conflits pouvant survenir durant l'exécution de l'application. Dans ce chapitre, nous présentons l'approche de placement mémoire sans conflit sous contrainte de réseau.

I. Introduction

Dans ce chapitre, nous introduisons une nouvelle approche d'allocation mémoire basée sur une heuristique gloutonne qui génère une architecture RTL d'un entrelaceur mémoire sans conflit à partir d'une règle d'entrelacement. Notre contribution comparée aux méthodes existantes consiste à considérer le réseau d'interconnexion comme étant une contrainte définie par le concepteur et non plus comme un objectif ciblé (i.e. [CHA10], [SAN11a]). L'approche que nous proposons peut être appliquée dans le cas des Turbo-Codes aussi bien que pour des codes LDPC, puisqu'elle est basée sur le concept « d'allocation mémoire double » (tiré de [CHA10a]) comme nous le verrons par la suite.

Notre approche, au-delà de se conformer aux désidératas du concepteur (définition du réseau d'interconnexion), vise également à optimiser le coût de l'architecture d'entrelacement dans son ensemble, c'est-à-dire en minimisant le coût de l'unité de contrôle associée. Pour rappel, l'architecture d'un entrelaceur mémoire (cf. Figure 3. 1) est généralement composée d'un réseau d'interconnexion reliant des unités de calculs à des bancs mémoire et d'une unité de contrôle dont une partie est dédiée pour le contrôle du réseau et une autre pour le contrôle des bancs mémoire. La problématique de l'optimisation de l'unité de contrôle sera toutefois développée dans le chapitre suivant.

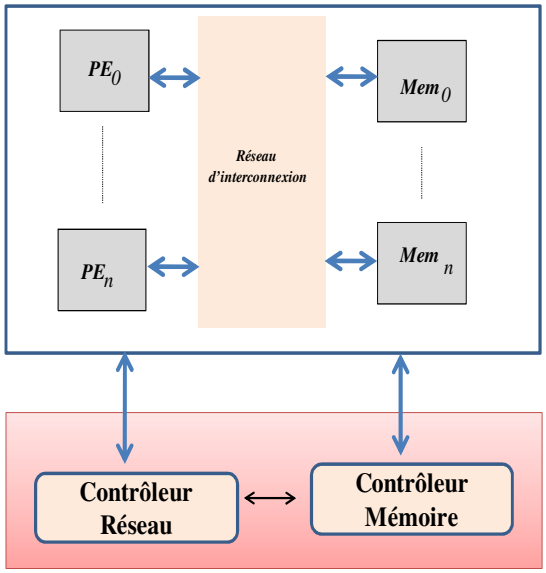


Figure 3. 1 Architecture typique d'un entrelaceur mémoire

Nous introduisons ainsi dans ce chapitre les concepts théoriques et les modèles formels ainsi que les différentes étapes détaillées de deux approches d'allocation mémoire permettant la conception d'une architecture d'entrelacement sans conflit. La première méthode cible l'optimisation de la partie réseau en explorant l'ensemble des solutions d'assignation des données aux bancs mémoire (tout en respectant une topologie réseau bien définie) de façon macroscopique en traitant d'un bloc l'ensemble de données mis en jeu à chaque cycle de codage/décodage. Cette première solution qui présente les principes clefs de notre approche à toutefois été largement améliorée pour aboutir à une seconde approche qui consiste à placer les données dans les bancs mémoire d'une manière plus fine en explorant les solutions de placement mémoire individuellement pour chaque accès aux données.

II. Formulation du problème d'allocation mémoire pour les Turbo-codes et les codes LDPC

Afin d'expliquer le problème, nous considérons un ensemble de structures de données $\{d_0,d_1,.....,d_{D-1}\}$ et un ensemble d'éléments de traitement $\{PE_0,PE_1,.....PE_{p-1}\}$ qui traitent les D structures de données en T cycles $\{t_0,t_1..., t_{T-1}\}$. Afin de stocker les D structures de données et réaliser un traitement itératif parallèle des données, nous utilisons un ensemble de B bancs mémoire $\{b_0b_1,....,b_{B-1}\}$ avec $B \geq P$.

Modèle

La problématique d'allocation mémoire est classiquement modélisée par une matrice appelée matrice d'accès aux données (e.g. Figure 3. 2). Cette matrice a P lignes qui correspondent aux P éléments de traitement et T colonnes qui représentent les instants t_i. Elle contient 6 données dont chacune est accédées 3 fois, et ce dans un ordre pseudo-aléatoire (défini par une règle d'entrelacement).

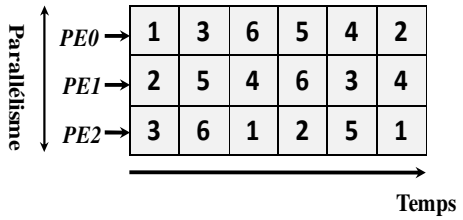
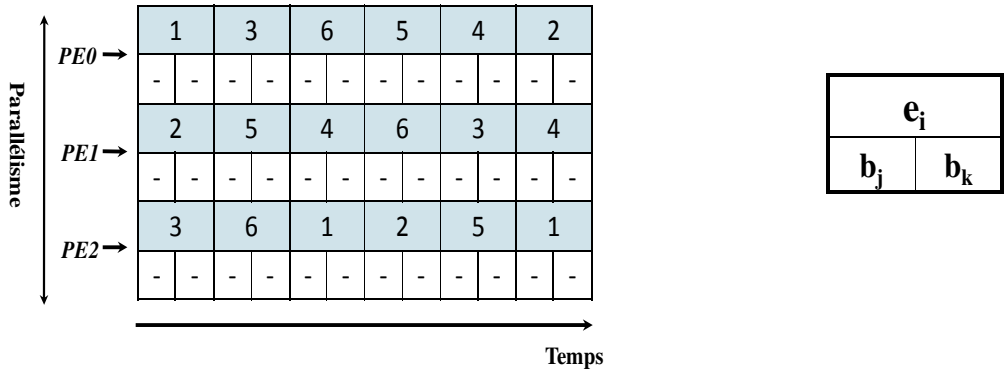


Figure 3. 2 *Matrice d'accès aux données*

Allocation mémoire double

Afin d'appliquer le concept d'« allocation mémoire double», chaque colonne est divisée en deux sous colonnes. La première sous colonne contient les bancs mémoire dans lesquels les données qui sont accédées en parallèle à l'instant t_i sont lues et la seconde sous colonne montre les bancs mémoire dans lesquels les résultats du traitement de ces données sont écrits. En outre, les données de chaque ligne sont accédées par le processeur qui est connecté à cette ligne. La Figure 3. 3.b illustre un élément de la matrice d'affectation où la donnée e_i est lue dans le banc mémoire b_j et écrite après avoir été traitée par un processeur dans le banc mémoire b_k .



Matrice d'affectation

Un élément de la matrice d'affectation

Figure 3. 3 *Matrice d'affectation pour la mémoire d'accès aux données*

Afin de faciliter les explications de nos approches nous définissons une semi-colonne de la manière suivante :

Définition Semi-colonne

On parle de **semi-colonne** de lecture (resp. d'écriture) pour désigner l'ensemble des accès en lecture (resp. en écriture) réalisés à un cycle de traitement t_i.

Contraintes d'assignation

Afin de garantir une allocation mémoire valide, un ensemble de contraintes de placement mémoire et des contraintes de réseau doivent être respectées.

1- Contraintes de placement mémoire :

- Les données qui sont traitées au même instant (lues par les éléments de traitement, ou bien écrites), i.e. qui appartiennent à une même semi-colonne dans la matrice d'affectation, doivent être assignées à des bancs mémoire différents.

- Le $n^{ème}$ accès en lecture de la donnée e_i doit être effectué dans le même banc mémoire que son $(n\text{-}1)^{ème}$ accès en écriture, c'est-à-dire la donnée doit être lue dans le même banc mémoire où elle a été écrite. En outre, puisque les algorithmes de décodage sont basés sur des approches itératives, si le premier accès en lecture à la donnée e_i est effectué dans le banc mémoire b_j alors le dernier accès en écriture à la donnée e_i doit être effectué dans ce même banc mémoire b_j.

2- Contraintes de réseau :

L'allocation mémoire doit respecter une topologie de réseau définie par l'utilisateur. Ceci se traduit par un ensemble de permutations de données que devra pouvoir réaliser l'architecture. Nous pouvons citer l'exemple d'un « Barrel Shifter » qui peut réaliser seulement des permutations circulaires tandis qu'un « Cross bar » peut effectuer toutes les permutations.

Exemple :

Soit les trois éléments suivants : A, B, C.
Alors,
- Les permutations réalisées par un Barrel Shifter sont : *ABC, CAB, BCA.*
- Les permutations réalisées par un Cross bar sont : *ABC, CAB, BCA, ACB, BAC, CBA.*

Problème d'assignation

Le problème à résoudre peut se formuler ainsi : « Stocker *D* données dans *B* bancs mémoire de telle sorte que les *P* éléments de traitement puissent accéder sans aucun conflit et en parallèle à chaque instant aux *P* bancs mémoire pour la lecture de ces *P* données et ensuite l'écriture des *P* résultats correspondant. »

III. Allocation mémoire sans conflit sous contrainte de réseau d'interconnexion

A. Première approche d'allocation mémoire

Nous proposons dans cette section une première méthode d'assignation mémoire permettant de résoudre les problèmes de conflits mémoire lors de la conception d'applications dédiées aux Turbo-Codes et aux codes LDPC. Son point de départ est une règle d'entrelacement modélisée par une matrice d'accès aux données, et une contrainte architecturale qui consiste à respecter une topologie d'un réseau particulier défini par le concepteur.

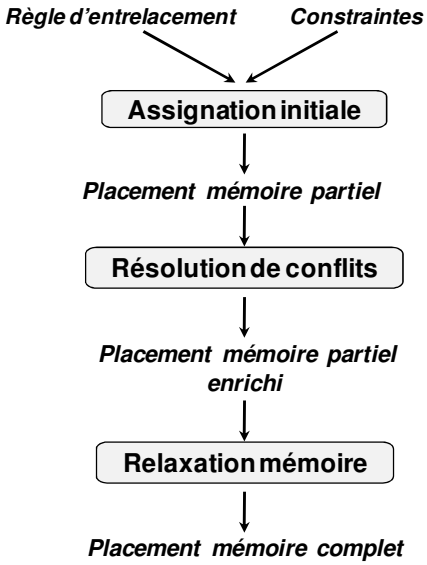


Figure 3. 4 *Flot de conception*

L'approche peut être divisée en deux étapes successives : un premier algorithme d'assignation mémoire est utilisé afin de générer une allocation partielle de la mémoire qui sera sans conflit mémoire et sans conflit réseau. Durant cette étape appelée « *Assignation Initiale* », (cf. Figure 3. 5), si un ensemble de données ne peut être assigné à un banc mémoire sans violer les contraintes de placement mémoire et de réseau, cet ensemble de données ne sera momentanément pas assigné à des bancs mémoires. Cette première étape génère ainsi une matrice d'allocation mémoire partiellement remplie.

Ensuite, la seconde étape de notre méthode, appelée étape de « *Résolution de Conflits* » va tenter de résoudre un maximum de ces conflits, sans modifier l'assignation précédente, ce que nous appelons un *placement mémoire partiel enrichi*.

Finalement, toutes les données non assignées sont mémorisées dans des registres additionnels, c'est ce que l'on appellera la « *Relaxation Mémoire* » (cf [BRI12a]). Ces registres servent à étendre l'espace mémoire disponible, à la différence de [THU03]

Définition *Relaxation mémoire*

On parle de **relaxation mémoire** lorsque l'on élargit l'espace mémoire disponible en lui rajoutant des éléments de mémorisation supplémentaires (registres, FIFO...).

Une fois la matrice d'affectation générée, une dernière étape consiste à minimiser le coût des éléments de mémorisation supplémentaires (i.e. les registres générés par l'étape de relaxation de la contrainte mémoire) et des multiplexeurs associés. Cette étape est réalisée grâce à l'approche STAR proposée dans la thèse de doctorat de C. Chavet [CHA07]. Cette approche, à partir de l'analyse des durées de vie d'un ensemble de données, optimise le coût des registres ainsi que celui-ci de la logique d'aiguillage nécessaire.

a) Assignation initiale

Cette première étape a pour objectif de générer une première allocation mémoire sans conflit. Si un ensemble de données accédées en parallèle ne respecte pas les contraintes de la mémoire ou bien du réseau, alors les assignations de ces éléments conflictuels sont reportées, les différentes étapes de l'assignation initiale sont illustrées dans la Figure 3. **5**.

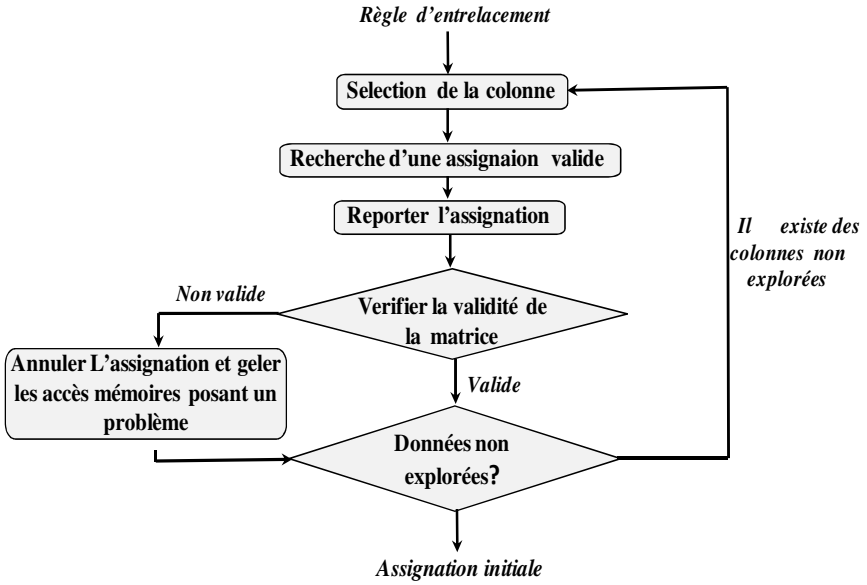


Figure 3. **5** *Assignation initiale*

L'assignation initiale est effectuée comme suit jusqu'à ce que toutes les colonnes aient étés traitées.

Sélection de la colonne :

Dans cette étape l'algorithme sélectionne la colonne la plus contrainte et qui n'a pas encore été explorée. La semi-colonne la plus contrainte (lecture ou écriture) est celle qui contient le nombre minimum de données n'ayant pas encore été assignées à des bancs mémoire.

Assignation et report :

L'algorithme recherche une assignation mémoire valide respectant les contraintes de placement mémoire et de réseau pour la semi-colonne sélectionnée. Il s'agit d'une solution valide localement à la semi-colonne sélectionnée. Ensuite, l'algorithme met à jour la matrice d'affectation en reportant l'assignation de cette semi-colonne sur les cellules concernées. La validité de ce report n'est pas vérifiée dans cette étape.

Vérification :

Dans cette étape, l'algorithme vérifie si les cellules qui ont été mises à jour dans l'étape « assignation et report » respectent les contraintes de mémoire et de réseau. Si ce n'est pas le cas, les assignations des données appartenant aux semi-colonnes qui ne respectent pas les

contraintes seront annulées et gelées. En effet, ces accès mémoire ne seront plus pris en compte au cours de cette étape d'assignation initiale.).

Nous introduisons certains termes qui sont utilisées dans la majorité des algorithmes que nous détaillerons par la suite :

C_i : Une semi-colonne de la matrice d'assignation.

Lc_i : Une solution d'assignation mémoire valide pour la colonne C_i.

d_k : une donnée de la colonne C_i.

Ld_k : une solution d'assignation mémoire valide pour la donnée d_k.

<table>
<tr>
<td>
Entrées :
Matrice d'allocations mémoires vide
Matrice d'accès aux données
Sorties :
Matrice d'allocation mémoire partiellement assignée
Début
Tant que toutes les colonnes C_i n'ont pas encore été explorées
Sélectionner la colonne la plus contrainte C_i
Générer une assignation mémoire localement valide Lc_i pour C_i
Affecter la solution Lc_i dans la colonne C_i
Reporter l'assignation dans les colonnes qui sont reliées à C_i
Vérifier la validité de la solution dans la matrice d'affectation
Si la solution n'est pas valide alors
Annuler les assignations non valides
Fin Si
Fin Tant que
Fin
</td>
<td>
Entrées :
Matrice d'allocations mémoires
Matrice d'accès aux données
Sorties :
Matrice d'allocations mémoires modifiée
Début
Pour toute colonne C_i dans la matrice M
Si la colonne C_i ne respecte pas les contraintes de réseau et de mémoire alors
Pour toute donnée d_k dans C_i
Si d_k a été déjà assignée à un banc mémoire alors
Annuler l'assignation de d_k
Fin Si
Fin Pour
Fin Si
Fin Pour
Fin
</td>
</tr>
<tr>
<td>a. Assignation initiale</td>
<td>b. Vérification</td>
</tr>
</table>

Figure 3. 6 *Algorithme de l'assignation initiale*

b) Résolution des conflits

Cette deuxième étape part d'une matrice partiellement affectée, résultante de l'assignation initiale. À chaque itération, l'algorithme sélectionne lui aussi la colonne la plus contrainte, ensuite pour chaque donnée il génère une solution d'assignation mémoire respectant les contraintes de réseau et de mémoire dans la matrice d'affectation.

La génération d'une solution d'allocation mémoire valide pour une donnée dans la matrice doit respecter la propriété de *consubstantialité* :

Soient,

M une matrice d'affectation mémoire
$C_i \epsilon M$ une semi colonne de lecture non assignée
$C_j \epsilon M$ une semi colonne d'écriture non assignée

d_k une donnée non assignée et qui est accédée en lecture dans le cycle *i* et accédée en écriture dans le cycle *j*
S_i l'ensemble de solutions localement valides pour d_k dans la colonne C_i
S_j l'ensemble de solutions localement valides pour d_k dans la colonne C_j

Propriété Consubstantialité

L'ensemble de solutions valides pour d_k dans la matrice d'affectation M est :

$$S_M(d_k) = \{S_i \cap S_j\}$$

```
Entrées :
        Matrice d'allocation mémoire partiellement assignée
        Matrice d'accès aux données
Sorties :
        Matrice d'allocation mémoire totalement assignée
Début
        Tant qu'il existe des colonnes vides ou partiellement
         assignées
           Sélectionner la colonne la plus contrainte Ci
           Pour tout dk non assignées dans Ci
              Générer l'ensemble de solutions valides à dk
              Si l'ensemble de solutions est vide alors
                 Affecter un registre à dk
              Sinon
                Affecter une solution valide à dk
              Fin Si
              Reporter l'assignation dans la colonne concernée
           Fin Pour
        Fin Tant que
Fin
```

Figure 3. 7 *Algorithme de résolution de conflits*

Si certaines de ces données n'ont pas de solution mémoire valide (i.e. $S_M(d_k) = \emptyset$) des registres supplémentaires sont mis en œuvre afin de stocker ces éléments conflictuels pour la relaxation de la contrainte mémoire. Il s'agit là d'un nouveau concept que nous introduisons dans le cadre de la conception d'entrelaceurs parallèles. Comme évoqué précédemment, le principe est d'accroitre l'espace d'exploration des solutions de placement mémoire. L'approche que nous proposons dans le cadre de cette thèse ne met pas en œuvre de rétroaction (i.e. recommencer l'ensemble du placement mémoire en utilisant le nouvel espace des solutions possibles).

c) Analyse de complexité

Cette première approche est ainsi basée sur une exploration de l'espace des solutions semi-colonne par semi-colonne. En résumé, après sélection de la semi-colonne la plus contrainte, l'approche détermine un placement mémoire pour toutes les données de celle-ci, reporte ce placement dans le reste de la matrice, puis vérifie qu'aucun conflit avec les contraintes de placement n'existe. Si c'est le cas, la semi-colonne est gelée. Enfin, lorsque toutes les semi-colonnes ont été explorées, l'étape de résolution va se charger de résoudre les conflits mis en attente via le gel des semi-colonnes.

Considérons le nombre de données à traiter *N* et le parallélisme de traitement de l'application *P*, cette première approche commence, comme nous venons de le rappeler, par définir une solution de placement mémoire pour chaque semi-colonne (soit $O(((N/P).(N/P +1))/2)$ étapes) si possible (sinon la semi-colonne ne sera plus explorée dans cette étape). Puis pour chaque semi-colonne dont les données n'ont pu être assignées à un banc mémoire (dans le pire cas $O(((N/P).(N/P +1))/2)$ semi-colonnes sont concernées) l'approche va sélectionner un placement mémoire valide pour le sous-ensemble de données de la semi-colonne qui ne présente pas de conflit avec les contraintes de placement, et ajouter un registre pour les accès aux données pour lesquels aucune solution respectant les contraintes de placement n'est possible. Il faut donc tester toutes les données non assignées de la colonne parcourue, soient dans le pire cas *P* données. Ainsi la complexité finale de cette seconde étape est $O(P.((N/P).(N/P +1))/2)=O((N.(N/P +1))/2)$.

Cette première approche « gloutonne » de placement mémoire a donc une complexité polynomiale : $O(((N/P).(N/P +1))/2) + ((N.(N/P +1))/2) = O((N/2).(N/P +1).(1/P +1)) \approx O(N^2)$. Le prix à payer de cette relative faible complexité tient au nombre de registres qui seront être ajoutés par l'étape de relaxation mémoire comme nous le verrons dans le chapitre 5.

B. Exemple pédagogique

Afin d'illustrer cette première approche de placement mémoire avec relaxation de la contrainte mémoire, nous proposons ici un exemple à visée pédagogique. Considérons l'exemple de la loi de permutation présentée par le modèle matricielle illustrée dans la dans la Figure 3. 2. L'architecture cible est ici composée de trois bancs mémoire (A, B, C) et trois unités de traitement (PE0, PE1, PE2), le réseau d'interconnexion ciblé est un Barrel Shifter. Intuitivement, on comprend qu'afin de pouvoir implémenter un tel réseau, chaque colonne de la matrice finale (après assignation mémoire) devra être une permutation circulaire de toutes les autres.

Basée sur le modèle matricielle illustré dans la Figure 3. 3, notre algorithme commence par faire une assignation des données de la première semi-colonne dans des bancs mémoire: Les accès en lecture aux données *1*, *2*, et *3* sont ainsi respectivement assignés aux bancs mémoire *A*, *B* et *C*.

Afin de respecter les contraintes de placement mémoire, le même banc mémoire doit être assigné au premier accès en lecture à une donnée ainsi qu'au dernier accès en écriture à cette même donnée. Les derniers accès en écriture de ces trois données sont donc assignés aux mêmes bancs mémoire, comme indiqué dans la Figure 3. 8 illustrant la mise à jour de la matrice d'affectation.

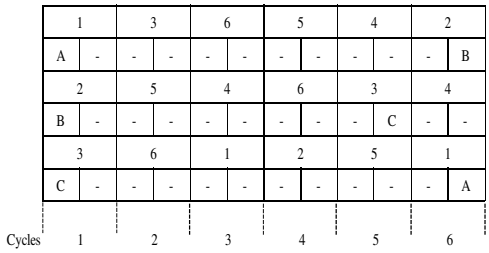
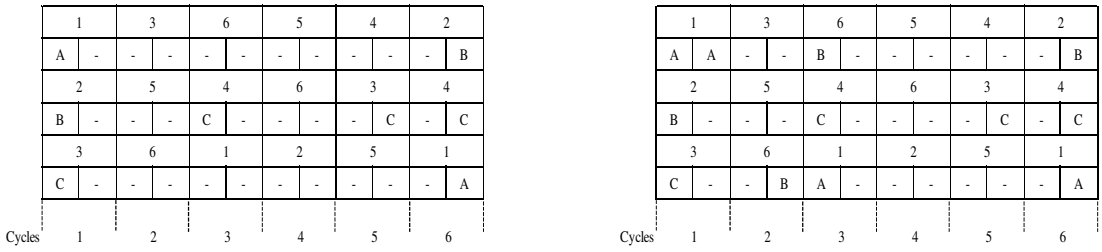


Figure 3. 8 *Report de l'assignation de la première colonne*

L'algorithme sélectionne ensuite la semi-colonne (lecture ou écriture) la plus contrainte. L'algorithme tente d'affecter les données non assignées dans la semi-colonne sélectionnée, tout en respectant les contraintes réseau et mémoire. Si l'on se réfère à la Figure 3. 8 la semi-colonne la plus contrainte est la dernière (accès en écriture au cycle 6). Ainsi, la donnée 4 est affectée au seul banc mémoire disponible : le banc C. A nouveau, afin de respecter la contrainte de mémoire, la cellule

correspondant au premier accès en lecture à la donnée 4 est mise à jour avec le même banc mémoire (banc C, cf Figure 3. 9.a).



(a) Deuxième assignation (Cycle 6)

(b) Troisième assignation (Cycle 3)

Figure 3. 9 *Deuxième et troisième assignation*

A ce moment, deux semi-colonnes peuvent être considérées comme les plus contraintes : la semi-colonne de lecture à l'instant 3 et celle d'écriture à l'instant 5. Dans ce cas l'algorithme sélectionne par défaut la semi-colonne de cycle le plus précoce, ce qui correspond à l'accès le plus ancien dans le temps. Dans notre exemple, l'algorithme sélectionne la semi-colonne de lecture au cycle 3. Afin de respecter la contrainte de réseau, ce qui est équivalent à suivre une permutation circulaire caractérisant le réseau d'interconnexion Barrel-Shifter, la donnée *6* et la donnée *1* sont respectivement assignées aux bancs mémoire *B* et *A*. Selon les contraintes de mémoire, les cellules qui correspondent à leurs précédents accès en écriture sont elles aussi assignées respectivement aux bancs mémoire *B* et *A*, comme illustré dans la Figure 3. 14.b. Notre algorithme va continuer à s'exécuter selon le même principe sur le reste de la matrice tant qu'aucun conflit d'assignation mémoire ne se révèle.

A l'instant 4, l'assignation des données *5* et *2* engendre une violation des contraintes du réseau. En effet, dans la semi-colonne d'écriture au cycle 2, la donnée *5* a été assignée au banc mémoire *A*, ce qui implique sa prochaine lecture (au cycle 4) doit être effectuée dans ce même banc mémoire. En outre, dans la semi-colonne d'écriture du cycle 1, la donnée *2* à été assignée au banc mémoire *B*, ce qui implique sa prochaine lecture doit être effectuée dans ce même banc mémoire *B*. Néanmoins, les accès en lecture aux données *5* et *2* au cycle 4 nécessiteraient une permutation des bancs mémoire qui ne respectent pas avec les contraintes du réseau ciblé (i.e. Barrel-Shifter). Ainsi, les solutions de placement mémoire s'offrant à nous ne permettent pas de respecter toutes les contraintes de placement. Par conséquent, les données *5* et *2* ne sont momentanément pas assignées à un banc mémoire. Cette première étape de placement mémoire ne pourra plus les prendre en compte, ces éléments sont donc rendue inaccessibles (représentés par les cellules grisées dans la Figure 3. 5.a. Tant que toutes les cellules de la matrice d'assignation n'ont pas étés explorées, cette première étape de placement mémoire se poursuit selon le même principe.

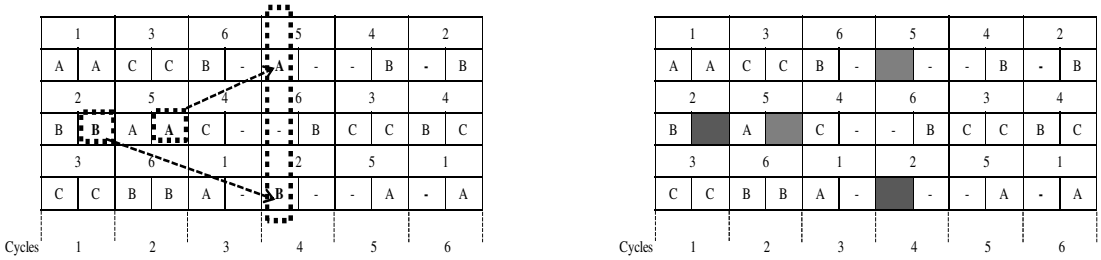


Figure 3. 10 *Echec de contrainte architecturale dans la colonne de lecture à l'instant 4*

La Figure 3. 11 montre le résultat de cette première étape de placement mémoire : la matrice est partiellement assignée. L'étape suivante de notre approche consiste, à partir de là, à trouver un

maximum de solutions de placements mémoire respectant nos contraintes avec les bancs mémoire existant, et lorsqu'aucune autre solution n'est possible, alors la/les donnée(s) sera (ou seront) placée(s) dans un/des registres additionnels.

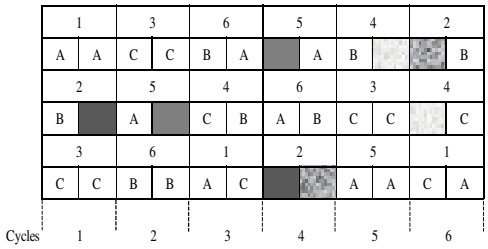


Figure 3. 11 *Matrice partiellement assignée résultat de l'étape initiale de l'algorithme*

C'est la seconde étape de notre algorithme qui se charge de trouver le placement mémoire final des données. Pour ce faire, l'algorithme détermine la semi-colonne la plus contrainte, puis pour chacune des données non assignées dans celle-ci on génère l'ensemble des solutions d'affectations mémoire valides (les contraintes de réseau et de placement mémoire). Lorsque cet ensemble de solutions est vide, la donnée est stockée dans un registre.

Dans notre exemple, la donnée *2* au cycle 1 de la matrice de la Figure 3. 11 possède une seule solution d'assignation respectant les contraintes (assignation au banc mémoire *B*), cependant la donnée *5* au cycle 2 n'a pas de solution d'affectation mémoire valide, car le banc mémoire *A* qui est une solution valide pour la donnée *5* dans la semi-colonne d'écriture au cycle 2 ne peut pas être utilisée dans la semi-colonne de lecture au cycle 4 (car dans ce cas la contrainte de réseau serait violée). Par conséquent, la donnée *5* est mémorisée dans un registre *R* entre le cycle 2 et le cycle 4. La seconde étape de notre algorithme s'exécute donc ensuite de la même façon jusqu'à ce que tous les accès aux données aient été assignés à un banc mémoire ou à un registre (cf Figure 3. 12).

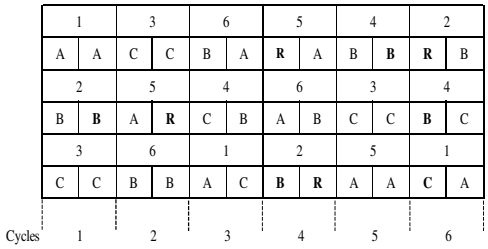


Figure 3. 12 *Solution valide d'assignation mémoire*

Une fois l'ensemble des données devant être stockées dans des registres défini, l'étape d'assignation des registres est mise en ouvre en générant une ensemble de contraintes pour le flot STAR [CHA07]. Dans notre exemple, puisque les durées de vie des données dans les registres ne se chevauchent pas, un seul registre additionnel est nécessaire pour pouvoir cibler un réseau d'interconnexion réalisant un Barrel-Shifter.

Le processeur PE0 accède en lecture soit au registre R soit aux bancs mémoire (via le réseau d'interconnexion), ainsi un multiplexeur est mis en œuvre afin de pouvoir gérer les accès de ce processeur à ces éléments de mémorisation. La présence d'un autre multiplexeur dont sa sortie est connectée à l'entrée du registre est également nécessaire puisque celui-ci est accédé en écriture par deux processeurs (PE1 et PE2) dans des cycles d'horloges différents. La Figure 3. 13 illustre l'architecture finale générée.

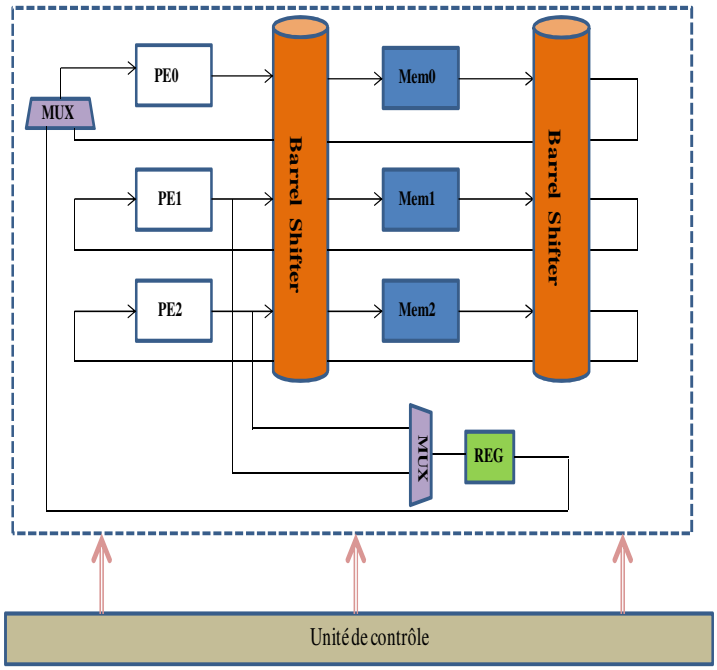


Figure 3. 13 *Architecture finale générée au niveau RTL*

Cet algorithme d'assignation mémoire tel que nous le décrivons ici est de complexité polynomiale, il permet à partir d'une règle d'entrelacement quelconque de générer une allocation mémoire sans conflit ciblant un réseau d'interconnexion défini par le concepteur. Ceci n'est possible que lorsqu'on s'autorise en contre partie à mémoriser certaines données dans des registres additionnels. Lors de l'étape d'*Assignation Initiale*, cette heuristique choisit une solution d'allocation mémoire valide pour la semi-colonne sélectionnée, néanmoins il s'agit d'une solution localement valide qui ne tient uniquement compte que de la semi-colonne en cour de traitement. Ceci a pour conséquence de potentiellement augmenter le nombre de données conflictuelles sensées d'être résolu dans l'étape suivante et complique ainsi la seconde étape de l'algorithme. En effet, notre heuristique traite une semi-colonne dans sa globalité sans tenir compte de l'ordre de traitement de ses données non assignées. Toutefois au sein d'une semi-colonne, certaines données sont plus contraintes que d'autres (une donnée peut avoir moins de solutions d'allocation mémoire valides qu'une autre).

C'est à partir de ces constatations que nous proposons de modifier notre approche en cherchant une assignation mémoire « donnée par donnée » plutôt que « semi-colonne par semi-colonne ». En outre, au lieu de chercher une solution locale pour la donnée sélectionnée, la nouvelle heuristique que nous proposons vise dans son étape d'*Assignation Initiale* à chercher une solution valide pour la donnée dans la matrice d'affectation. Dans le cas où une telle solution n'existe pas, elle se contente d'une solution localement valide.

IV. Allocation mémoire avec sélection de la donnée la plus contrainte

La méthode que nous proposons ici s'articule comme précédemment autour de deux étapes principales d'exploration : l'étape d'« Assignation Initiale » et l'étape de « Résolutions des Conflits » que nous allons étudier.

A. Nouvelle approche

a) Assignation initiale

Cette première étape a pour objectif de générer une première allocation mémoire sans conflit. Si un ensemble de données accédées en parallèle ne respecte pas les contraintes de la mémoire ou bien du réseau, alors les assignations de ces éléments conflictuels sont reportées, les différentes étapes de l'assignation initiale sont illustrées dans (cf. Figure 3. 14).

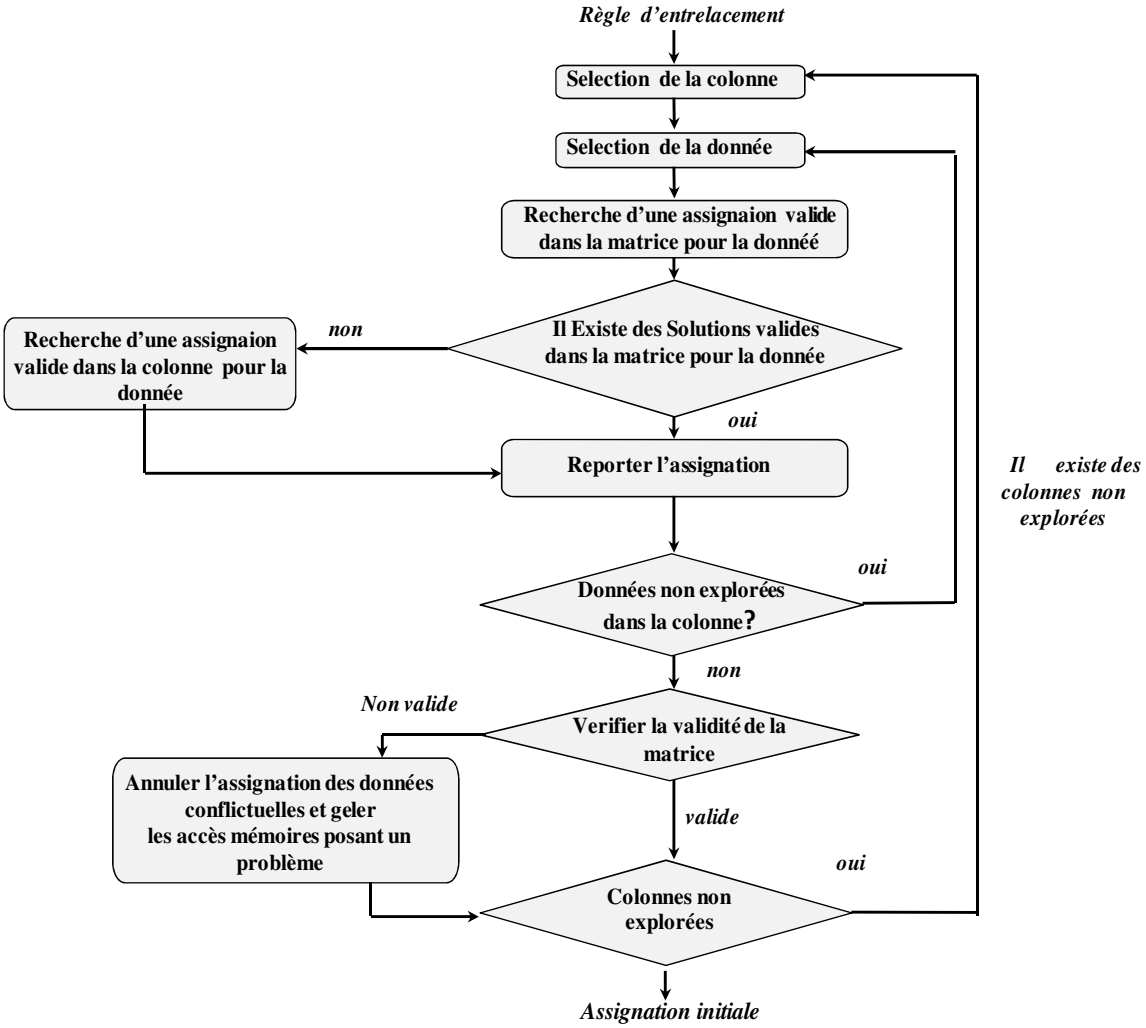


Figure 3. 14 *Sélection « donnée par donnée »dans l'assignation initiale*

L'assignation initiale est effectuée comme suit jusqu'à ce que toutes les colonnes aient étés traitées.

Sélection de la colonne :

Dans cette étape l'algorithme sélectionne la colonne la plus contrainte et qui n'a pas encore été explorée. La semi-colonne la plus contrainte (lecture ou écriture) est celle qui contient le nombre minimum de données n'ayant pas encore été assignées à des bancs mémoire.

Assignation et report :

L'algorithme recherche une assignation mémoire valide donnée par donnée, en sélectionnant à chaque itération la donnée la plus contrainte (celle ayant le nombre minimal, non nul, de solutions valides dans la matrice d'affectation). Une solution d'assignation d'une donnée est valide dans la matrice si elle peut être appliquée aux deux accès de la donnée (lecture et écriture, cf. propriété de consubstantialité). Si une telle solution n'existe pas, l'algorithme se contente dans cette étape d'une solution d'assignation valide pour la semi-colonne sélectionnée. Ensuite, l'algorithme reporte l'affectation de la donnée dans la cellule concernée afin de respecter les contraintes de placement mémoire.

Entrées :

Matrice d'allocation mémoire vide
Matrice d'accès aux données

Sorties :

Matrice d'allocation mémoire partiellement assignée

Début

Tant que toutes les semi-colonnes n'ont pas encore été explorées dans M
Sélectionner la semi-colonne C_i la plus contrainte dans M
Sélectionner la donnée d_k la plus contrainte dans la semi-colonne C_i
Générer les solutions Ld_k d'assignation mémoire valides pour la donnée d_k
Si l'ensemble de solutions d'assignations mémoire de d_k valide dans la matrice est vide **alors**
Générer une solution d'assignation mémoire valide uniquement dans la semi-colonne C_i
Fin Si
Affecter une solution s_k de Ld_k générée pour la donnée d_k
Reporter la solution s_k dans les semi-colonnes qui lui sont reliées.
Vérifier la validité de la solution dans le reste de la matrice
Si La solution n'est pas valide dans une des semi-colonnes contentant un accès à une donnée relié à d_k **alors**
Annuler les assignations non valides
Fin Si
Fin Tant que

Fin

Figure 3. 15 *Algorithme d'assignation initiale traitant en priorité la donnée la plus contrainte*

Vérification :

Dans cette étape, l'algorithme vérifie si les cellules qui ont été mises à jour dans l'étape « assignation et report » respectent les contraintes de mémoire et de réseau. Si ce n'est pas le cas, les assignations des données appartenant aux semi-colonnes qui ne respectent pas les contraintes seront annulées et gelées. En effet, ces accès mémoire ne seront plus pris en compte au cours de cette étape d'assignation initiale.

b) Résolution des conflits

Dans cette étape, plutôt que de traiter l'ensemble de données conflictuelles semi-colonne par semi-colonne, l'algorithme résout les conflits « donnée par donnée », en sélectionnant à chaque itération la

donnée la plus contrainte (donc pas nécessairement dans la même colonne que le dernière donnée traitée). Les données qui n'ont pas de solutions d'allocation mémoires valides seront stockées dans des registres, toujours selon notre principe de relaxation mémoire.

```
Entrées :
        Matrice d'allocation mémoire partiellement assignée
        Matrice d'accès aux données
Sorties :
        Matrice d'allocation mémoire totalement assignée
Début
        Tant qu'Il existe des données non assignées dans M
            Sélectionner la donnée d_k la plus contrainte dans M
            Générer une solution valide Ld_k à d_k
            Si l'ensemble de solutions valide est vide alors
                Affecter un registre à d_k
            Sinon
                Affecter Ld_k à d_k
            Fin Si
            Reporter l'assignation dans la cellule concernée
        Fin Tant que
Fin
```

Figure 3. 16 *Résolution des conflits « donnée par donnée »*

Cette heuristique est une approche gloutonne de complexité polynomiale, elle permet de générer une allocation mémoire sans conflit et de cibler un réseau d'interconnexion particulier. En outre, elle cherche à minimiser le nombre des registres alloués en favorisant à chaque itération le traitement des données les plus contraintes.

c) Analyse de complexité

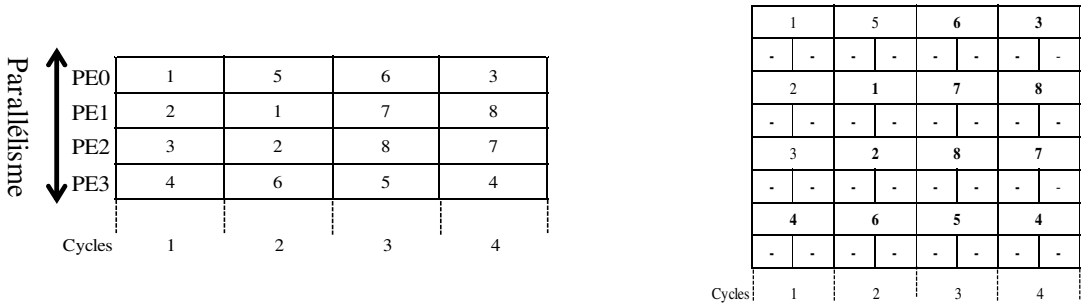
Dans cette seconde approche l'espace des solutions est explorer en déterminant le placement mémoire des données une par une. En résumé, après sélection de la semi-colonne la plus contrainte, l'approche détermine un placement mémoire pour sa donnée la plus contrainte, reporte ce placement dans le reste de la matrice, puis vérifie qu'aucun conflit avec les contraintes de placement n'existe. Si c'est le cas, l'accès à la donnée est gelée. Enfin, lorsque toutes les données ont été explorées, l'étape de résolution va se charger de résoudre les conflits mis en attente via le gel des données.

Notre méthode commence ici par chercher les accès aux données les plus contraints, et définit une solution de placement mémoire pour la donnée sélectionnée (soit $O(((N.(N/P +1))/2) . ((P+1)/2))$ étapes). Puis (étape de résolution de conflit), pour chaque donnée non assignée à un banc mémoire l'approche va sélectionner un placement mémoire valide pour le sous-ensemble de données de la semi-colonne concernée qui ne présente pas de conflit avec les contraintes de placement (i.e. $O((N.(N/P +1))/2)$), et ajouter un registre pour les accès aux données pour lesquels aucune solution respectant les contraintes de placement n'est possible.

La complexité de cet algorithme en terme de nombre d'étape de calcul pour le placement mémoire est $O(((N.(N/P +1))/2) . ((P+1)/2) + (N.(N/P +1))/2)) \approx O(N^2 + N^2/P)$.

B. Exemple visée pédagogique

Pour mettre en exergue l'approche que nous proposons, de la façon la plus pédagogique possible nous proposons un nouvel exemple : On considère la règle d'entrelacement représentée par la matrice d'accès aux données illustrées dans la Figure 3. 17.a. Notre architecture cible est composée par 4 bancs mémoire (*A, B, C, D*), 4 processeurs (*PE0, PE1, PE2, PE3*) et un réseau d'interconnexion cible de type papillon afin de mettre plus clairement en exergue les particularités de cette seconde approche.



a. Matrice d'accès aux données b. Matrice d'assignation mémoire

Figure 3. 17 *Matrices relatives à l'exemple pédagogique*

Les contraintes d'assignation

De nouveau, afin de garantir une allocation mémoire valide, des contraintes mémoires et réseau doivent être respectées. Toutefois, la contrainte de réseau que nous utiliserons ici sera différente puisque nous souhaitons cette fois cibler un réseau d'interconnexion de type papillon.

Un papillon de parallélisme 4 contient deux étages ; chaque étage contient 2 commutateurs 2x2 (cf. Figure 3. 18).L'ensemble de permutations possibles d'un papillon à 4 entrées est 2*4 =16 permutations :

Etant données 4 bancs mémoire *{A, B, C, D}* : l'ensemble des permutations possibles effectuées par un réseau papillon à 4 entrées = *{ACBD, ACDB, CABD, CADB, ADBC, ADCB, DABC, DACB, BCAD, BCDA, CBAD, CBDA, BDAC, BDCA, DBAC, DBCA}*.

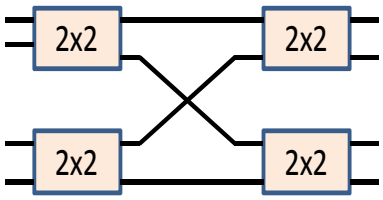


Figure 3. 18 *Réseau d'interconnexion papillon avec un parallélisme de 4 à base de papillon élémentaire de parallélisme 2*

Placement mémoire

La matrice d'assignation mémoire (cf Figure 3. 17.a), correspond à la matrice d'accès aux données de la Figure 3. 17.b. L'algorithme commence par effectuer une assignation de la première semi-colonne (au cycle 1), en respectant les contraintes de réseau et de mémoire. Ainsi, les accès en écriture

aux données *1, 2 ,3* et *4* sont affectées respectivement aux bancs mémoire *A, C, B, D* (voir Figure 3. 19).

1		5		**6**		**3**	
-	A	-	-	-	-	B	-
2		1		7		8	
-	C	A	-	-	-	-	-
3		2		8		7	
-	B	C	-	-	-	-	-
4		6		5		4	
-	D	-	-	-	-	D	-
Cycles	1		2		3		4

Figure 3. 19 *Affectation de la première colonne d'écriture*

Ensuite, l'algorithme sélectionne la semi-colonne la plus contrainte (e.g. ici deux choix son possibles, la semi-colonne de lecture du cycle 2 ou la semi-colonne de lecture du cycle 4). Supposons que l'algorithme sélectionne la semi-colonne de lecture du cycle 2. L'algorithme génère l'ensemble de solutions d'allocations mémoire valides pour chacune des données non assignées et choisit celle qui a le nombre minimale de solutions valides et non vide. Dans notre exemple, les données non assignées dans la semi-colonne de lecture du cycle 2 sont les données *5* et *6*.

L'accès en écriture de la donnée *6* est effectué au cycle 3, la semi-colonne d'écriture du cycle 3 étant vide, ainsi l'accès en écriture de la donnée *6* peut être effectué dans n'importe quel banc mémoire (*A, B, C* ou *D*). Néanmoins, l'accès en lecture correspondant est effectué au cycle 2. Afin de cibler une permutation papillon, la seule solution valide pour l'accès en lecture de la donnée *6* est le banc mémoire *B*. De la même façon la donnée *5* a une seule solution de placement mémoire valide : le banc mémoire *D*. En conclusion, les deux données *5* et *6* ont une unique solution de mémoire valide, ils ont ainsi le même degré de contrainte, dans ce cas l'algorithme choisit celle qui est traitée par défaut par le processeur d'indice le plus petit ; ainsi la donnée *5* qui est traitée par le processeur PE_0 sera sélectionnée (cf. Figure 3. 20).

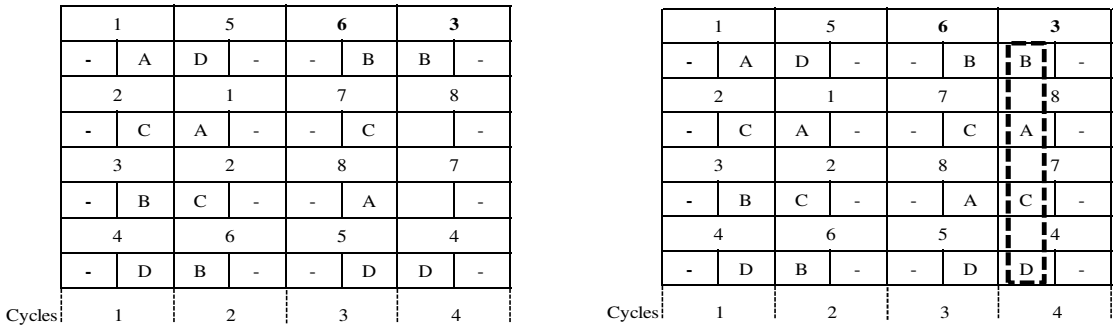
1		5		**6**		**3**	
-	A	D	-	-	B	B	-
2		1		7		8	
-	C	A	-	-	-	-	-
3		2		8		7	
-	B	C	-	-	-	-	-
4		6		5		4	
-	D	B	-	-	D	D	-
Cycles	1		2		3		4

Figure 3. 20 *Assignation de la deuxième colonne*

Ensuite, l'algorithme sélectionne la semi-colonne d'écriture la plus contrainte. Ici deux semi-colonnes peuvent être sélectionnées (semi-colonne d'écriture au cycle 3 et semi-colonne de lecture au cycle 4). Supposons que l'algorithme sélectionne la semi-colonne d'écriture du cycle 3 : les données non assignées dans cette colonne sont ainsi *7* et *8*.

Afin d'avoir une permutation possible par un réseau papillon, l'accès en écriture de la donnée *7* (semi-colonne du cycle 3) doit être effectué dans le banc mémoire *C* ; cependant son accès en lecture (semi-colonne du cycle 4) doit être réalisé dans le banc mémoire *A*. Par conséquent il n'y a pas de solution mémoire valide pour la donnée *7*.

De même, on constate que l'ensemble de solutions d'assignation mémoires valides pour la donnée *8* est vide. Dans une telle situation, l'algorithme trouvera une solution valide par rapport à la colonne sélectionnée, ce qui correspond dans notre exemple à une solution valide pour les données *7* et *8* pour leur accès en écriture au troisième cycle : la donnée *7* est affectée au banc mémoire *C* et la donnée *8* au banc mémoire *A* (voir Figure 3. 21. a).



a. Assignation de la troisième colonne

b. Conflit d'accès et violation des contraintes de réseaux

Figure 3. 21 *Violation des contraintes de réseaux*

Dans la phase de vérification, l'algorithme détecte une violation de contraintes, qui apparait dans la semi-colonne de lecture du cycle *4* (cf. Figure 3. 22.b), la permutation ainsi réalisée ne respecte pas un réseau de type papillon. Dans ce cas, l'algorithme reporte l'assignation de cette colonne à la deuxième étape.

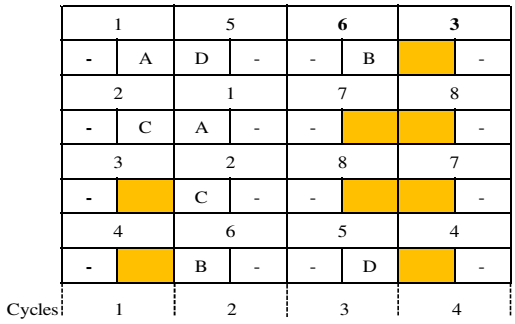


Figure 3. 22 *Annulation de l'assignation des données conflictuelles*

De même, l'algorithme va chercher à assigner les bancs mémoire aux autres données (cf Figure 3. 15). En définitive, la Figure 3. 23 illustre le résultat de l'étape d'assignation initiale : les différents accès aux données sont assignés à des bancs mémoires, excepté les accès ne respectant pas les contraintes de placement mémoire que nous avons définies.

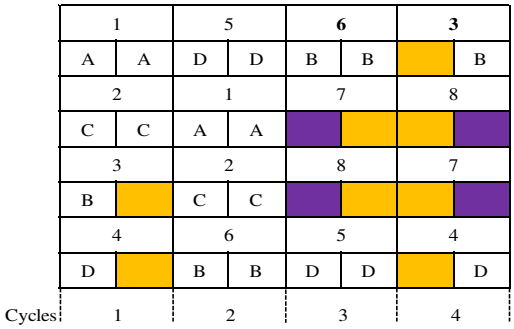


Figure 3. 23 *Résultat de l'assignation initiale*

La seconde étape de l'algorithme va alors s'atteler à la résolution des conflits restant. A chaque itération, l'algorithme sélectionne l'accès à une donnée non assignée à un banc mémoire le plus contraint. Dans notre exemple, les accès aux données conflictuelles non assignées sont : *3, 4, 7 et 8*. Les données *3* et *4* ont deux solutions d'assignations mémoires valides {*D*, *B*}, tandis que les données *7* et *8* de la semi-colonne de lecture du cycle 3 ont respectivement une seule solution d'allocation mémoire possible C et A.

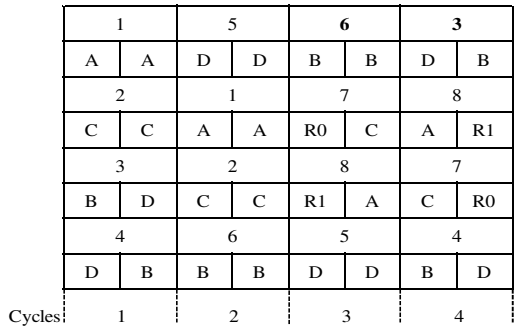


Figure 3. 24 *Résultat final d'allocation mémoire sans conflits*

Néanmoins les données *7* et *8* de la colonne d'écriture du cycle 4 n'ont pas de solutions d'assignation mémoire valides. Par conséquent, l'algorithme commence par assigner les données *7* et *8* de la semi-colonne d'écriture du cycle 3, puis les données *3* et *4*. Enfin, puisque les données *7* et *8* de la semi-colonne d'écriture du cycle 4 n'ont pas de solutions d'allocation mémoire valides, elles seront stockées dans des registres additionnels. La Figure 3. 24 illustre le résultat final de notre algorithme.

Finalement deux registres supplémentaires R0 et R1 sont mis en œuvre afin de garantir l'utilisation possible d'un réseau d'interconnexion de type « papillon ». L'accès en lecture de ces deux éléments de mémorisation est réalisé respectivement par les deux processeurs PE1 et PE2, ainsi on a également besoin de deux multiplexeurs afin de gérer les accès de ces processeurs (cf. Figure 3. 25).

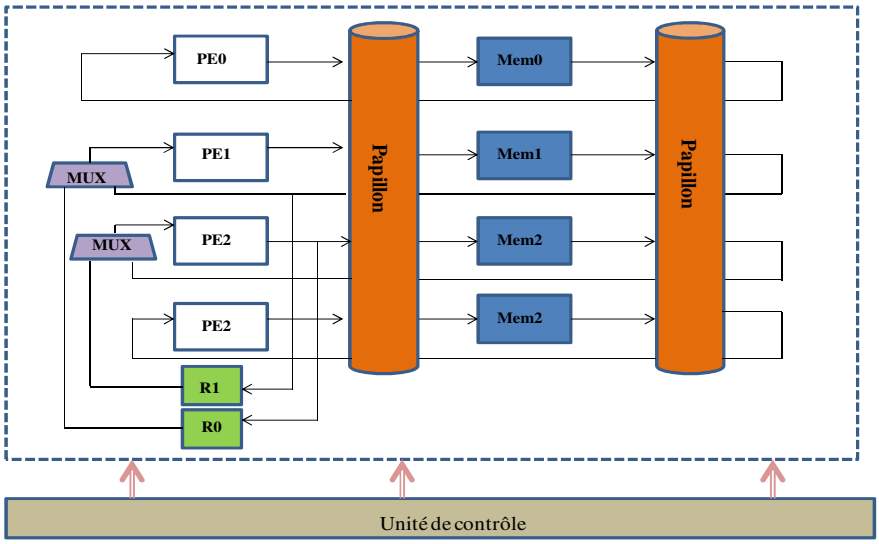


Figure 3. 25 *Architecture RTL générée*

V. Conclusion

Dans une première partie de ce chapitre, nous avons présenté une approche d'allocation mémoire permettant de générer une architecture RTL d'un entrelaceur mémoire sans conflit à partir d'une règle d'entrelacement. Contrairement aux techniques existantes, cette méthode est capable de systématiquement cibler un réseau d'interconnexion défini par l'utilisateur pour n'importe quelles lois de permutation. Elle utilise dans sa résolution du problème d'assignation mémoire une heuristique constructive, et utilise des registres pour la mémorisation des données violant les contraintes. Cette approche est appelée *Relaxation mémoire*. En outre, l'intérêt de cette approche devient plus évident si le degré de parallélisme est élevé puisque, dans ce cas, le coût d'un réseau d'interconnexion augmente considérablement (Pour un réseau de Benes, le nombre de commutateurs est $(N.log_2N) - N/2$ avec N est le degré de parallélisme).

Dans une seconde partie, nous avons proposé une approche d'allocation mémoire également sans conflit et respectant une contrainte de réseau définie par l'utilisateur. Cette approche se base sur une exploration plus fine de l'espace des solutions pour obtenir une solution utilisant moins de registre additionnels (i.e. issus de la relaxation de la contrainte mémoire). Nous présentons dans le chapitre 5 une analyse détaillée des performances respectives de ces approches.

Avoir une solution d'assignation mémoire sans conflit et ciblant un réseau d'interconnexion particulier est toujours possible grâce à notre approche puisqu'elle autorise la mémorisation des données conflictuelles dans des registres. Par conséquent, notre problème devient plutôt un problème d'optimisation dont la fonction objectif est de minimiser le nombre de registres pour que le coût de ceux-ci ne soit pas trop important. Nous sommes toutefois là face à un problème NP-complet, et un résultat optimal n'est pas toujours garanti, surtout si la loi de permutation est complètement incompatible avec le réseau d'interconnexion ciblé. Dans un tel cas, une estimation et fixation d'un seuil maximal de registres à ne pas dépasser s'avère nécessaire afin de déterminer l'intérêt de cibler un réseau particulier. Parmi les solutions que nous pouvons envisager, il serait possible de tester plusieurs réseaux d'interconnexion cibles et déterminer ensuite celui qui présente un bon compromis entre coût de réseau ciblé et coût de registres. Ces approches seront succinctement décrites par la suite,

puisqu'elles font l'objet de travaux en cours dans le cadre d'une collaboration avec un autre doctorant du laboratoire (Saeed Ur Reehman).

Toutefois comme nous l'avons évoqué dans le chapitre précédent, au delà de la problématique de placement de données en mémoire sous contrainte de réseau d'interconnexion, un autre axe d'optimisation important concerne le coût du contrôleur de l'architecture. Dans le chapitre suivant, nous proposons ainsi un niveau d'analyse plus fin de l'assignation mémoire qui consistera à placer les données en mémoire de façon à minimiser le coût de ce contrôleur.

Chapitre 4
Placement mémoire et optimisation du contrôle

Sommaire

Dans le chapitre précédent, nous avons présenté deux approches de placement mémoire sans conflit sous contrainte de réseau. Nous savons toutefois qu'une seconde problématique doit être traitée : l'optimisation du coût de l'unité de contrôle de l'architecture. Ce chapitre présente l'approche formelle que nous proposons pour tenter de répondre à cette question.

I. Introduction

Le placement de données en mémoire et l'optimisation de l'architecture résultante que nous proposons se base sur des modèles formels et des approches exploratoires que nous avons présentées dans le chapitre précédent. Ces modèles sont exploités par un flot de synthèse dédié à la génération d'architectures d'entrelacement de données (Réseau d'interconnexion, Mémoires et Unité de contrôle) de niveau transfert de registres (RTL, pour Register Transfer Level). Toutefois, afin de pouvoir optimiser le coût architectural du contrôleur, nous devons modifier quelque peu ces algorithmes, et proposer de nouveaux outils théoriques.

Dans un premier temps nous proposons une description détaillée de l'architecture que nous ciblons, puis nous introduirons les aspects théoriques que nous allons devoir mettre en œuvre. Nous utiliserons quelques exemples à visée pédagogique afin de montrer les apports des solutions que nous proposons dans ces travaux de thèse. Enfin nous exposerons les évolutions possibles de nos solutions dans un cadre d'utilisation plus général.

Rappels détaillés de l'architecture cible

Le chemin de données de l'entrelaceur mémoire que nous ciblons (cf. Figure4. 1) est composé de processeurs communiquant en parallèle avec des bancs mémoire via un réseau d'interconnexion défini par le concepteur, des éléments de mémorisation supplémentaires permettant de mémoriser transitoirement les données conflictuelles, de la logique d'aiguillage afin de pouvoir assurer le partage de ces éléments entre les différents processeurs.

Le pilotage des mémoires pourrait être réalisé soit avec une machine à états finis qui détermine, à chaque cycle d'horloge, à quelle adresse le processeur accède à la mémoire, soit en effectuant un prétraitement hors ligne qui consiste à mémoriser les mots de commandes dans des ROMs dédiées. Il est alors nécessaire d'avoir au minimum une ROM par banc mémoire afin de garantir un traitement parallèle des données. Réalisé soit par une machine à état fini ou bien par des ROMs, la complexité du contrôleur mémoire augmente considérablement avec la taille de la trame de données traitées par chaque processeur et avec le degré du parallélisme de l'entrelaceur. Afin de pouvoir réduire cette complexité nous proposons un contrôleur mémoire basée sur des ROMs et nous introduisons dans ce contexte, dans une deuxième partie de ce chapitre une approche permettant d'effectuer un placement intelligent des données dans la mémoire ciblant l'optimisation de la taille et le nombre des ROMs utilisées.

La partie contrôle peut être divisée en trois unités : une unité qui est dédiée pour le contrôle du Réseau, les mots de commandes de celle-ci sont stockées dans une ROM qui est elle-même pilotée par un compteur. Une deuxième unité permettant le contrôle des registres ainsi que les multiplexeurs qui leurs sont associés. Et enfin une troisième unité dédiée au contrôle des bancs mémoire (également basé sur un ensemble de ROMs). Les ROMs mémorisant les séquences de mot de commande sont optimisées afin d'en minimiser leurs tailles. Naturellement une logique d'aiguillage dédiée est alors nécessaire afin de distribuer les signaux de contrôle en sortie des ROMs vers les RAMs. Chaque ROM d'adressage peut alors être contrôlée par son propre compteur permettant d'incrémenter l'adresse de lecture dans la ROM à chaque fois que celle-ci est parcourue.

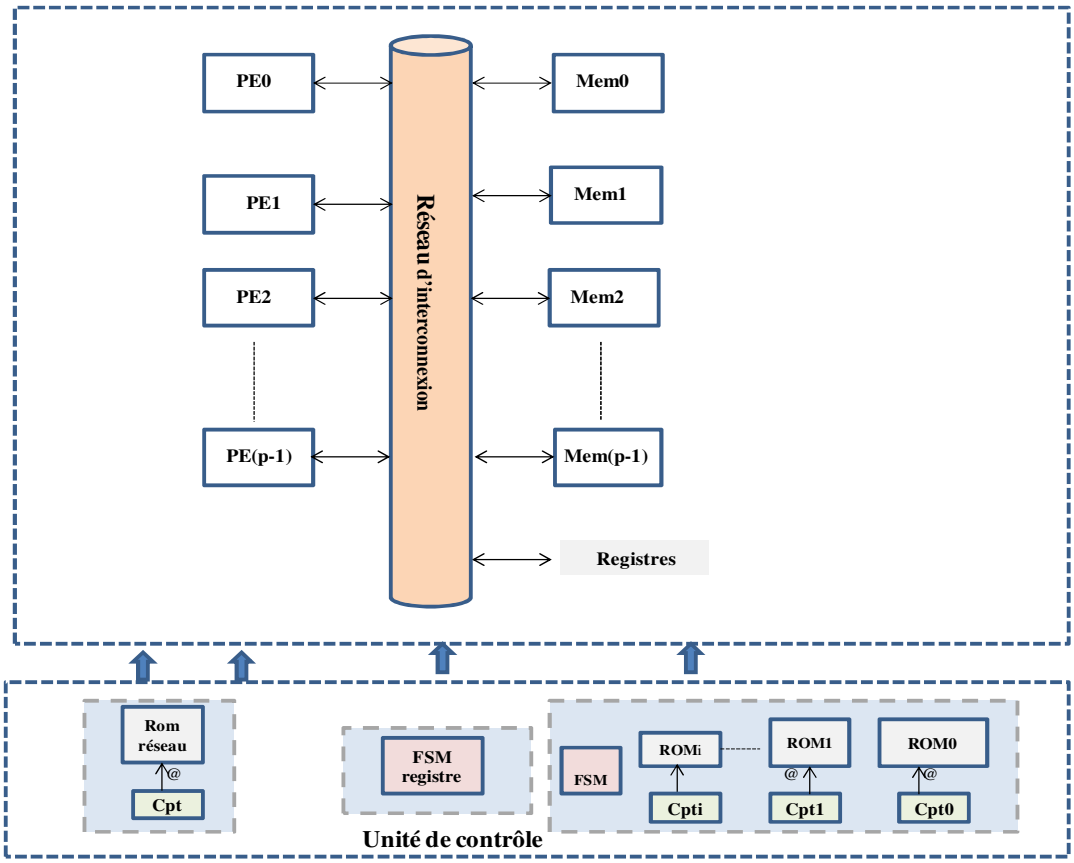


Figure4. 1 *Architecture cible d'un entrelaceur mémoire parallèle*

C'est dans ce contexte que nous proposons l'automatisation de la génération et de l'optimisation de l'adressage mémoire. L'objectif est de pouvoir réduire la taille et le nombre des différentes ROMs utilisées. En effet, en essayant de placer les données dans la mémoire de manière intelligente, i.e. en permettant à plusieurs processeurs d'accéder dans le même cycle à la même adresse dans les différents bancs mémoire, il n'est plus nécessaire de mémoriser cette adresse dans toutes les ROMs. Dans ce cas il suffit qu'elle soit stockée dans une seule ROM et elle sera partagée par les différentes RAMs. Ainsi, la difficulté consiste à extraire une certaine régularité au niveau de placement des données dans les RAMs, autrement dit à maximiser l'occurrence d'une adresse mémoire qui sera mémorisée dans les différentes ROMs et lu par les RAMs dans un même cycle d'horloge. Plus cette régularité sera élevée, plus les tailles des ROMs seront réduites. Au final, nous aurons minimisé le nombre total de ROMs, celles-ci étant partagées par toutes ou partie des RAMs, et donc le coût du controleur associé.

Comme nous le verrons par la suite, l'optimisation de la surface du contrôleur mémoire peut également réduire d'une manière considérable le coût matériel de l'entrelaceur, cependant dans la littérature, les concepteurs n'accordent leur attention qu'à l'optimisation du réseau d'interconnexion et au placement des données dans les bancs mémoires.

II. Placement des données en mémoire sans conflit pour la génération d'un contrôleur mémoire optimisé

Compte tenu du modèle architectural visé, la problématique de l'optimisation du contrôle consistera dans un premier temps, à faire en sorte qu'à chaque cycle d'accès aux données, les adresses de lecture ou d'écriture soient les mêmes pour tous les bancs mémoires, dans la mesure du possible. Ainsi, il sera possible par la suite de fusionner les structures contenant les mots de commandes destinés à chaque

banc mémoire, au sein d'une seule et même structure. Pour atteindre cet objectif, la première étape consistera à formaliser les accès aux adresses mémoire par un modèle de graphe spécifique.

A. Graphe de Conflit d'Adresse

Dans le cadre de cette thèse, nous proposons un nouveau modèle de graphe de conflit destiné à la formalisation des conflits entre les accès aux adresses des données en mémoire : le Graphe de Conflit d'Adresse (ou ACG pour Address Conflict Graph). En premier lieu, nous introduisons certaines terminologies nécessaires pour la présentation du modèle formel.

Définition Durée de vie d'une donnée

Soient

$T_{écr}(d_i)$= temps d'écriture de d_i par un processeur dans un banc mémoire B_i.

$T_{lec}(d_i)$= temps de lecture de d_i par un processeur à partir du banc mémoire B_i.

La durée de vie est alors définie par l'intervalle temporel

$\Delta_{di} = [t_{écr}(d_i), t_{lec}(d_i)]$

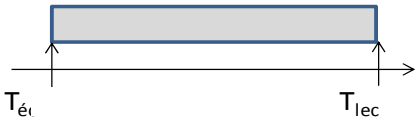


Figure 4. 2 *Durée de vie d'une donnée*

Définition Chevauchement de durées de vie

Soient

Δ_{di} = durée de vie d'une donnée d_i.

Δ_{dj} = durée de vie d'une donnée d_j.

Deux données ont des durées de vie Δ_{di} et Δ_{dj} qui se chevauchent ssi,

$\Delta_{di} \cap \Delta_{dj} \neq \emptyset$

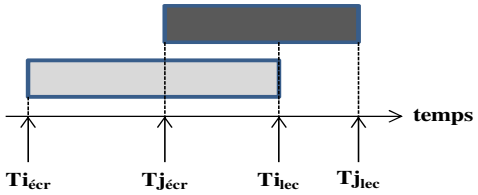


Figure4. 3 *Chevauchement de durée de vie de deux données*

Définition d'un Graphe de Conflit d'Adresse

Un **graphe de conflit d'adresse**, ou ACG (pour Address Conflict Graph), est un graphe de contraintes non orienté G(V,E) :

- L'ensemble des nœuds du graphe **V**, contenant les nœuds représentant les accès aux données.

- L'ensemble des arcs de compatibilité E = {(v_i, v_j)} représente l'existence d'un conflit d'adresse entre les nœuds v_i et v_j (i.e. deux adresses différentes doivent être utilisée pour les nœuds v_i et v_j)

Exemple :

Soient d_1, d_2, d_3 *et* d_4 des données avec

d_1 et d_2 assignées au banc b_i,

d_3, d_4 assignées au banc b_j.

Les durées de vie sont illustrées dans la Figure4. 4 et le graphe de conflits d'adresses correspondant est présenté dans la Figure4. 5.

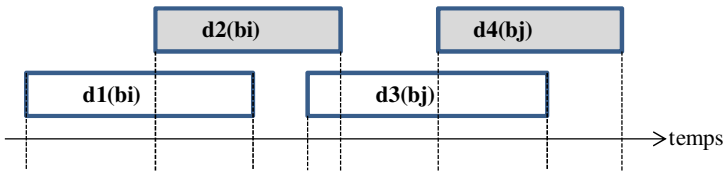


Figure4. 4 *Durées de vie de l'ensemble des données (d_1, d_2, d_3, d_4)*

Ce graphe de conflits d'adresses est composé de deux sous graphes, un associé au banc mémoire b_j et le second associé au banc mémoire b_i.

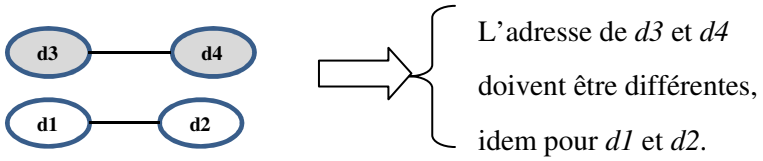


Figure4. 5 *Graphe de conflit d'adresse associés aux données (d_1, d_2, d_3, d_4)*

B. Exploitation d'un ACG pour l'optimisation de l'architecture de contrôle

Comme nous venons de la voir, l'assignation de deux données dans un même banc mémoire peut créer un conflit d'adresse, selon les durées de vie de ces données dans ce banc mémoire. De ce fait, nous constatons que la manière avec laquelle les données sont assignées aux bancs mémoires a un impact sur la « complexité » du graphe de conflits d'adresses résultant. Le nombre d'arcs de conflits et le degré de chaque nœud appartenant au graphe sont des conséquences directes de l'assignation des données aux bancs mémoires (cf [BRI13a]).

a) Génération d'un Graphe de Conflits d'Adresses

Pour obtenir un ACG, nous allons modifier l'algorithme d'assignation mémoire « donnée par donnée » proposé précédemment afin de définir un nouvel objectif que devront prendre en compte les étapes d'Assignation Initiale et de Résolution des Conflits. Ainsi, le placement des données en

mémoire se fera en respectant les contraintes de placement mémoire et de réseau précédemment définies, mais également avec un objectif qui sera de minimiser la complexité du graphe ACG généré en parallèle. Cette minimisation se fera lors de la sélection du banc mémoire à assigner aux données en élisant le banc qui créera le moins d'arcs de conflit dans le graphe ACG. Cette étape se fera par l'intégration d'une heuristique dédiée au cours de l'étape de placement mémoire. Le flot de conception que nous proposons (cf. Figure4. 6 Flot de conception pour la génération de l'entrelaceur mémoire parallèle) permet à partir d'une règle d'entrelacement et d'une contrainte de réseau (i.e. le réseau que souhaite cibler le concepteur) de générer une architecture optimisée (VHDL de niveau RTL), d'un entrelaceur mémoire.

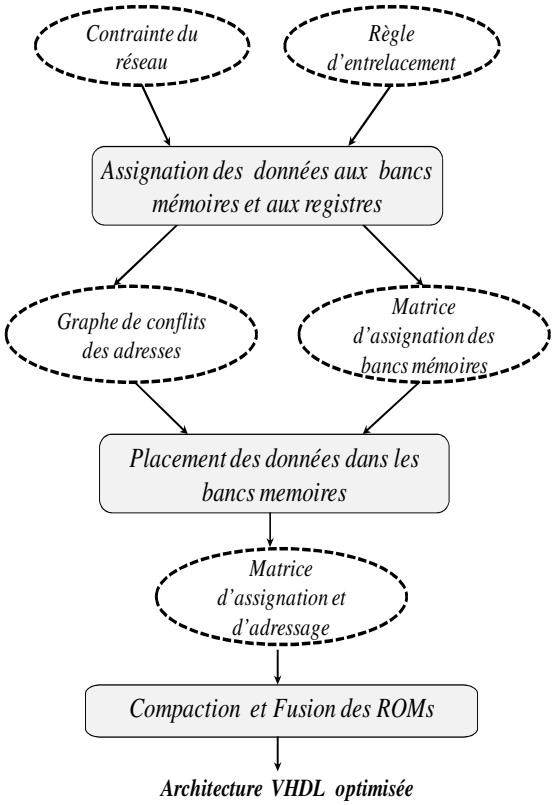


Figure4. 6 *Flot de conception pour la génération de l'entrelaceur mémoire parallèle*

La première étape consiste à assigner les données aux bancs mémoires et aux registres, tout en générant le graphe ACG. Ainsi, après cette étape d'Assignation des données, nous obtenons (1) un graphe de conflit d'adresse et (2) une matrice d'assignation des données dans les bancs mémoires. Ces deux « modèles » serviront dans la seconde étape de notre flot à placer les données à leurs adresses dans les bancs mémoires.

Nota :

Dans la Figure4. 6, les étapes *Résolution de conflit* et *Relaxation mémoire* de la figure 3.4 ont été fusionnées dans le bloc *Placement des données dans les bancs mémoires*. En effet, ces deux étapes n'ont pas étés modifiées dans le cadre de la mise au point de notre nouvelle approche de placement mémoire avec optimisation de l'architecture de contrôle.

Enfin, à partir de la matrice d'assignation et d'adressage, une dernière étape d'optimisation permet de tirer partie de la régularité insérée dans les séquences d'adressage pour générer une unité de

contrôle fortement optimisée. Nous détaillons par la suite l'implémentation des étapes de ce flot. Nous explorons les solutions que nous avons présentées.

b) Assignation Initiale

Comme nous l'avons mentionné, l'approche de placement mémoire que nous proposons dans ce chapitre est basée sur l'approche d'allocation mémoire « donnée par donnée » d écrite dans le chapitre 3. Toutefois, plutôt que sélectionner la donnée la plus contrainte (i.e. la donnée ayant le minimum de solutions d'allocation mémoire valides), l'algorithme choisit le banc ayant le plus grand nombre de conflits d'adresses et lui affecte la donnée créant le minimum de conflits dans le ACG.

Afin de minimiser les conflits des adresses dans chaque banc mémoire, à chaque itération l'algorithme traite en priorité le banc mémoire dont le graphe ACG lui correspondant contient le plus grand nombre de conflits d'adresses. Il lui associe l'arête de poids le plus faible, ce qui permet d'équilibrer les conflits entre les différents bancs mémoires.

Ensuite le graphe biparti est mis à jour en éliminant toutes les arêtes violant la validité de la solution en cour de construction. En effet, si une arête sélectionnée relie un banc b_i et une donnée d_j, la mise à jour consiste à éliminer d'une part toutes les autres arêtes issues de b_i ou de d_i, ainsi que toutes les arêtes violant la contrainte de réseau. Ce processus est ainsi répété jusqu'au traitement de toutes les données accédées à l'instant courant t_k.

Etant donné un ensemble de données $\{d_0,\ldots d_{p-1}\}$ accédées simultanément par P processeurs à l'instant courant t_k, nous associons une fonction de coût à chaque solution d'allocation mémoire S_i affectant l'ensemble de données à l'ensemble de bancs mémoires $\{b_0,\ldots..b_{p-1}\}$. S_i représente un banc mémoire possible. Cette fonction de coût exprime le nombre de conflits des adresses créés suite à cette assignation (i.e. le nombre d'arcs de conflit créé dans l'ACG correspondant au banc mémoire S_i).

Coût d'assignation d'une donnée à un banc mémoire

- Soient Δd_i et Δd_j deux intervalles de durées de vie associées, respectivement, aux données d_i et d_j.
- Soit le poids W_{ij} associé aux deux données d_i et d_j, défini de la manière suivante

$$si\ \ \Delta d_i \cap \Delta d_j \neq \emptyset\ \ alors\ \ W_{ij}(d_i\ ,\ d_j) = 1$$
$$Sinon \qquad\qquad W_{ij}(d_i\ ,\ d_k) = 0$$

Le c**oût d'assignation d'une donnée** d_i à un banc b_j, noté par C_i correspond au nombre d'arcs de conflits crées suite à l'affectation de la donnée d_i au banc mémoire b_j.

$$C_i = \sum_{dk \in bj} W_{ik}\ (d_i, d_k) \qquad 4.1$$

Coût de la solution S_i

$$Coût(Si) = \sum_{k=0}^{p-1} C_k \qquad 4.2$$

A partir de l'ensemble des coûts d'assignation de chacun des bancs respectant les contraintes de placement mémoire, nous proposons une heuristique avec pour objectif de minimiser les conflits d'adresses par banc mémoire en se basant sur cette fonction de coût.

Pour ce faire, nous modélisons le problème par un graphe biparti *G=(D,B,E)* :

D est l'ensemble des P données à assigner dans le cycle courant t_k.
B est l'ensemble des bancs mémoires disponibles.
E est l'ensemble des arêtes pondérées.

La Figure4. 7 illustre un exemple de graphe biparti (cf. [SAN10]) pondéré avec $\{d_0,....d_{p-1}\}$ qui représentent les nœuds des *P* données (pouvant être accédées en même temps au sein d'une semi-colonne) et $\{b_0,...b_{p-1}\}$ qui représentent les nœuds de bancs mémoire. Une arête existe entre un nœud de banc mémoire et un nœud de donnée, si le banc mémoire fait parti de l'ensemble de solutions d'allocation mémoire valides respectant les contraintes de placement mémoire et de réseau. Le poids associé à chaque arête représente le coût C_i défini précédemment. Notre heuristique cherche une solution *S* valide minimisant la fonction de coût(S_i).

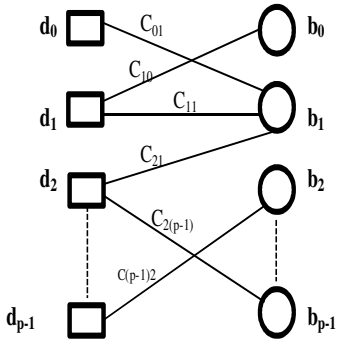


Figure4. 7 Modélisation du problème d'allocation mémoire dans un cycle particulier par un graphe biparti

Nota :

Une solution *S* est valide si elle respecte les contraintes de placement mémoire et de réseau imposées par le concepteur.

$$Coût(S) = min\big(Coût(Si)\big) = min\sum_{k=0}^{p} C_k \qquad 4.\ 3$$

Cette heuristique est appliquée pour chaque ensemble de données non assignées et accédées simultanément par les différents processeurs soit en écriture, soit en lecture. Dans la phase de vérification les affectations qui ne respectent pas les contraintes dans la matrice sont supprimées et les données en question ne peuvent plus être assignées aux bancs mémoires durant cette étape. Par la suite le ACG est mis à jour.

Entrées :

Graphe biparti pondéré G=(D,B,E) .
Graphe de conflit d'adresse (ACG)...

Sorties :

Un ensemble d'arrêtes pondérés.
Graphe de conflit d'adresse mis à jour.

Début

Pour chaque nœud b_i du G
Sélectionner le nœud du banc mémoire b_i caractérisé par le sous graphe ACG ayant le maximum de conflits.
Assigner au nœud b_i l'arête $e(b_i, d_i)$ de poids minimale avec e € E.
Mette à jour le graphe biparti G (Suppression de toutes les arêtes violant les contraintes).
Fin Pour
Mettre à jour le ACG.

Fin

Figure4. 8 *Algorithme d'assignation initiale tenant compte de la génération de ACG*

c) Phase de Résolution de Conflits

Durant la phase de résolution des conflits, le modèle est légèrement différent. En effet, une solution d'allocation mémoire valide n'exige pas nécessairement d'avoir assigné tous les bancs mémoires ou bien toutes les données puisque l'algorithme s'autorise à mémoriser certaines données dans des registres. Le principe de notre heuristique reste donc le même que ce que nous venons de décrire, à la différence près qu'une fois tous les bancs mémoire explorés, et en cas de présence de données n'ayant pas de solution d'allocation mémoire valide (i.e. équivalent à des nœuds de données non reliées à des arêtes) des registres seront mis en œuvre afin de mémoriser les données non assignées à des bancs mémoire.

Entrées :

Graphe biparti pondéré G= (D, B, E).
Graphe de conflit des adresses (ACG).

Sorties :

Un ensemble d'arrêtes pondérées et un ensemble de données assignées à des registres.
Graphe de conflit d'adresse mis à jour.

Début

Tant que tous les nœuds de bancs mémoires n'ont pas encore étés explorées
Sélectionner le nœud de banc b_i ayant le maximum de conflits et relié au moins à une arête.
Assigner au nœud b_i l'arête $e(d_i,b_i)$ de poids minimal.
Mette à jour le graphe G (Suppression de toutes les arêtes violant les contraintes).
Fin Tant que
Mettre à jour le graphe de conflit des adresses ACG.
Affecter des registres aux restes des données non assignées à des bancs mémoires.

Fin

Figure4. 9 *Résolution des conflits dans un cycle particulier*

d) Adressage des données au sein des bancs mémoire

Nous introduisons dans cette partie une approche automatisant le placement des données dans la mémoire (i.e. à quelle adresse mémoire sera placée la donnée) à partir d'un graphe de conflit d'adresse et d'une matrice d'assignation mémoire.

Entrées :

Matrice d'allocations mémoires
Graphe de conflits d'adresse

Sorties :

Matrice d'adressage

Début

Pour chaque nœud n_i du ACG
Sélectionner le nœud de degré de saturation maximal.
Générer l'ensemble des adresses valides pour n_i.
Sélectionner la solution maximisant la régularité d'adressage.
Fin Pour

Fin

Figure4. 10 *Algorithme d'adressage*

Pour ce faire, nous proposons une approche exploitant le graphe de conflits d'adresse et la matrice d'assignation mémoire. Il est important de noter que l'étape de placement mémoire décrite précédemment avec pour objectif la minimisation de la complexité des graphes ACG de chaque banc mémoire, mais ne vise pas en elle-même à minimiser la taille des ROMs. L'idée est plutôt d'offrir un espace de solution plus large pour l'étape d'adressage des données dans leur banc que nous allons maintenant décrire.

A chaque itération, l'algorithme sélectionne une donnée non assignée à une adresse mémoire, génère l'ensemble de solutions possibles pour cette donnée et enfin choisit la solution maximisant la fonction de coût. Le nœud sélectionné est le nœud le plus contraint dans le graphe de conflit d'adresse, on parle ici de *degré de saturation* d'un noeud.

Soient,
- *ACG(V,E)* un graphe de conflit d'adresse
- n_i un nœud du graphe de conflit

Définition Degré de saturation

Le **degré de saturation** de n_i = Nombre des nœuds qui ont déjà été assignés à leurs adresses mémoire et qui sont connectés à n_i dans ACG (V,E)
+
Degré de n_i dans ACG(V,E).

Une *solution d'adressage valide* est une solution ne violant pas les contraintes imposées par le graphe de conflit d'adresse : deux nœuds reliés par un arc dans le ACG doivent être affectés à deux

adresses mémoire différentes. Nous associons une fonction de coût à chaque solution d'adressage possible afin de favoriser d'une façon heuristique la régularité au niveau des séquences d'adressage mémoire.

Plus concrètement, la fonction de coût associé à une solution d'assignation d'une adresse $@_jk$ (adresse k dans le banc mémoire j) à une donnée n_i est composée par deux termes :

- le premier «*NbrOcc*» exprime le nombre de données déjà assignées à l'adresse $@_fk$ (avec $f\epsilon[b_0..b_{P-1}]$ et $f\neq j$)et qui sont accédées simultanément avec la donnée n_i (voir4. 4.a). Puisque la donnée est écrite et puis lue dans deux cycles d'horloge différents, le terme «*NbrOcc»* correspond aux accès à l'adresse $@_jk$ des différents bancs mémoires effectués dans ces deux cycles d'horloges (voir 4. 4.c).

Soient,

- n_i est un nœud de donnée dans le ACG.
- $@_jk$ est une adresse mémoire valide pour la donnée n_i.
- $@_fk$ est une adresse mémoire avec $f\epsilon[b_0..b_{P-1}]$ et $f\neq j$
- T_accès (n_i) correspond aux temps d'accès à la donnée n_i.
- T_accès_Conf (n_i) correspond aux temps des accès à toutes les données non assignées à des adresses mémoires et qui sont conflictuelles à n_i.
- T_lecture (n_i) correspond au temps de lecture de n_i.
- T_écriture (n_i) correspond au temps d'écriture de n_i.
- T_écriture_Conf (n_i) correspond aux temps d'écriture des données conflictuelles à n_i
- T_lecture_Conf (n_i) correspond aux temps de lecture des données conflictuelles à n_i.
- NbrOcc($@_fk$,T_accès(n_i))correspond au nombre des accès à l'adresse mémoire $@_fk$ effectué dans les cycles T_accès (n_i).
- MoyOccConf ($@_fk$, T_accès_Conf (n_i)) correspond à la moyenne des accès à l'adresse mémoire $@_fk$ effectué dans les cycles T_accès_Conf (n_i).
- MoyOccConf($@_fk$,T_écriture_Conf(n_i)) correspond à la moyenne des accès à l'adresse mémoire $@_fk$ effectué dans les cycles T_lecture_Conf (n_i).
- MoyOccConf($@_fk$,T_écriture_Conf(n_i)) correspond à la moyenne des accès à l'adresse mémoire $@_fk$ effectué dans les cycles T_écriture_Conf (n_i).

$$W(n_i,@_jk) = NbrOcc\ (@_fk,\ T\_accès\ (n_i)) - MoyOccConf\ (@_fk,\ T\_accès\_Conf\ (n_i)) \qquad 4.4\ (a)$$

$$MoyOccConf\ (@_fk,\ T\_accès\_Conf\ (n_i)) = MoyOccConf(@_fk,T\_écriture\_Conf(n_i)) \qquad 4.4\ (b)$$

$$+$$

$$MoyOccConf(@_fk,T\_lecture\_Conf(n_i))$$

$$NbrOcc(@_fk,T\_accès(n_i)) = NbrOcc(@_fk,T\_écriture\ (n_i)) + NbrOcc(@_fk\ ,T\_lecture\ (n_i)) \qquad 4.4\ (c)$$

Exemple :

Soit la matrice d'adressage mémoire partiellement assignée illustrée dans la Figure4. 11soient (A,B,C,D) les bancs mémoire disponibles.

Afin de calculer le « *NbrOcc »* associé à la solution d'assignation de $@_0A$ au troisième accès de la donnée 5, et puisque celle-ci est écrite dans le cycle 5 et puis lue dans le cycle 3,

nous considérons toutes les données assignées à l'adresses 0 dans les bancs mémoire différents de A et accédées soit en écriture dans le cycle 5, soit en lecture dans le cycle 3.

Dans notre cas, l'accès en écriture au cycle 5 de la donnée 6 est assigné à l'adresse 0 du banc B noté par $@_B0$, et l'accès en lecture au cycle 3 de la de la donnée 1 est assigné à $@_D0$.

Par conséquent ce «*NbrOcc*» est égale à 2.

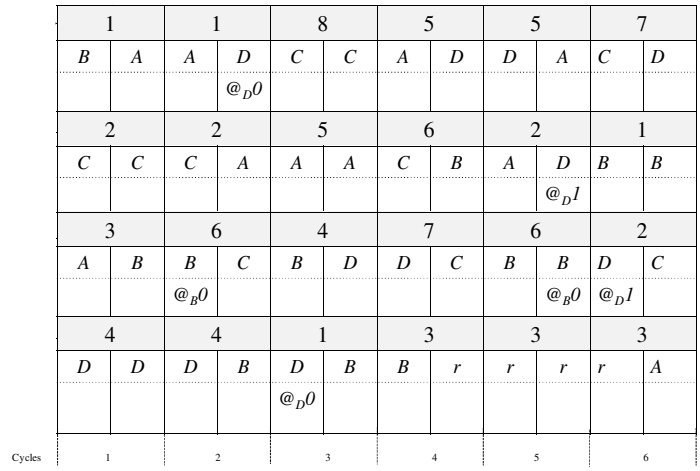


Figure4. 11 *Exemple d'une matrice d'adressage mémoire*

Néanmoins un nœud sélectionné a probablement des nœuds qui lui sont conflictuels dans le ACG, donc le choix d'une adresse particulière pour ce nœud va nécessairement interdire aux nœud qui lui sont relié d'utiliser cette même adresse. Ainsi le second terme «*MoyOccConf* » cherche à modéliser la régularité potentielle ainsi perdue pour la séquence d'adressage. Plus concrètement, ce deuxième terme exprime une valeur moyenne sur le nombre des données déjà assignées à l'adresse $@_jk$ et accédées simultanément en lecture et en écriture avec les données conflictuelles à n_i dans son ACG (voir 4. 4.b).

e) Analyse de complexité

L'approche que nous proposons ici est basée sur une exploration de l'espace des solutions donnée par donnée. Comme nous l'avons évoqué le placement des données dans les bancs mémoire se fait avec pour objectif de minimiser la complexité du graphe ACG qui est construit en parallèle. Considérons le nombre de données à traiter *N* et le parallélisme de traitement de l'application *P*, l'étape d'assignation initiale commence donc par chercher les accès aux données les plus contraints, et définit une solution de placement mémoire pour la donnée sélectionnée (soit $O(((N.(N/P +1))/2) . ((P+1)/2)$ étapes). Cette solution de placement sera, parmi toutes les solutions de placement possible qui respectent les contraintes, celle qui minimise la complexité du graphe ACG résultant. La sélection la dite solution de placement se fait en explorant chaque nœud des ACGs, donc dans le pire cas il faudra $O(N)$ étapes. Puis, l'étape de résolution des conflit avec relaxation mémoire reste la même que précédemment, i.e. $O((N.(N/P +1))/2)$. Enfin l'étape d'optimisation des ROMs commence par trier les différentes ROMs ($O(P)$), puis l'algorithme va fusionner les ROMs $O((N.P).(P-1)/4)$ cycles de traitement.

La complexité finale de l'approche s'exprime en $O(N.(N/P +1)/2 [(P+1)/2 + 1] + N.P.[(P-1)/4]) \approx O(N^2.P + N. P^2)$.

C. Exemple à visée pédagogique

Ainsi que nous l'avons présenté dans le chapitre précédent, l'approche que nous proposons ne se limite pas à trouver un placement de données en mémoires sans conflit et respectant un réseau d'interconnexion ciblé. En effet, cette approche vise également à optimiser le coût architectural de l'unité de contrôle du système. Pour mettre en exergue l'approche que nous proposons, de la façon la plus pédagogique possible nous proposons un nouvel exemple : le modèle matricielle suivant illustre les séquences d'accès aux données (cf. Figure4. 12). L'architecture cible est composée de 4 processeurs (*PE_0, PE_1, PE_2, PE_3*) communiquant avec 4 bancs mémoire (*A, B, C, D*) via un réseau d'interconnexion type papillon.

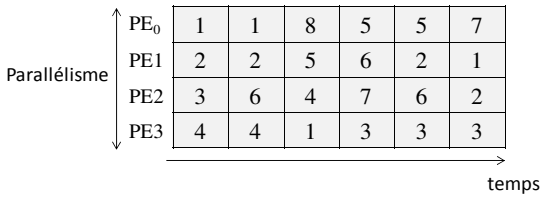
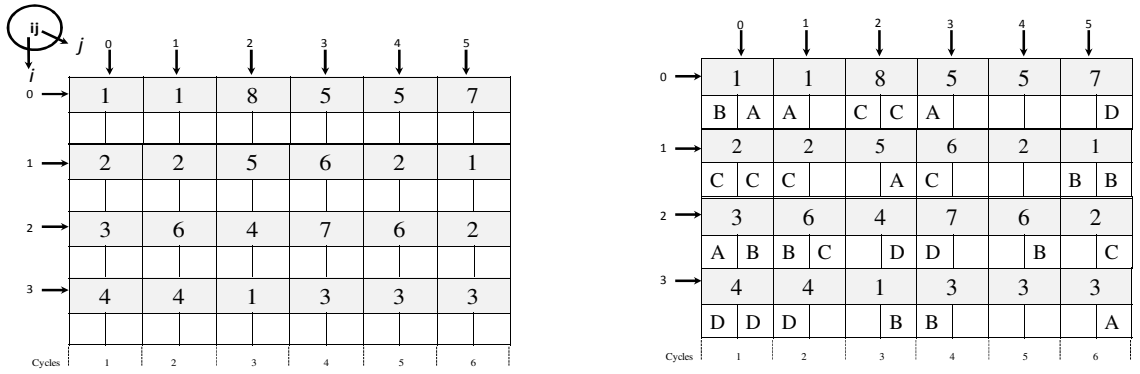


Figure4. 12 *Matrice d'accès aux données*

L'objectif de la première étape du flot est de générer l'assignation des données aux bancs mémoire tout en respectant les contraintes de mémoire et les contraintes de réseau (type papillon). Nous appliquons notre approche dans sa globalité, c'est-à-dire en prenant en compte la génération d'un graphe de conflit d'adresses. La Figure4. 13.a illustre la matrice d'assignation mémoire initiale. Afin de faciliter les explications qui suivent, nous avons choisi d'exprimer les données à travers leurs cordonnées dans la matrice d'assignation notées par *M(i,j)*, (cf. Figure 4.11.a) Ceci nous permet d'identifier individuellement et sans ambigüité les différents accès à une donnée.

Comme nous avons déjà décrit d'une manière détaillée l'algorithme d'assignation mémoire dans les deux exemples précédents, nous allons nous intéresser dans cette partie uniquement à une étape de l'assignation qui permet de mettre en avant la génération de l'*ACG.* Par conséquent, nous n'allons pas partir du début de l'algorithme mais plutôt après avoir exécuté quelques étapes de l'*Assignation initiale.*



(a) Matrice initiale

(b) Matrice partiellement affectée après quelques itérations

Figure4. 13 *Résultat d'affectation après quelques itérations*

Supposons qu'après quelques itérations nous obtenons la matrice M illustrée dans la Figure4. 13.b et le graphe de conflit d'adresse illustré dans la Figure4. 14. En observant le sous graphe ACG associé au banc mémoire *C (ACG_c)*, on peut déjà constater que la donnée 8 de cordonnée *M(0,2)* est en conflit

avec la donnée 6 de cordonnée *M(2,1)* et la donnée 2 de cordonnée *M(1,0)*. Ces conflits sont modélisés par deux arcs de conflit dans ACG_C reliant respectivement *M(0,2)* et *M(2,1)*, et pour le second *M(0,2)* et *M(1,0)*.

Les poids associés au graphe biparti sont calculés en utilisant la formule 3.2 détaillée dans le chapitre précédent. Si nous prenons l'exemple de *W (M(1,1), B)*, en analysant les durées de vie de l'accès à la donnée *M(1,1)* et des données qui sont assignées au banc mémoire B (cf. Figure4. 16), l'assignation de *M(1,1)* au banc mémoire *B* crée deux conflits d'adresse, ainsi le poids associé à cette assignation est égal à *2*.

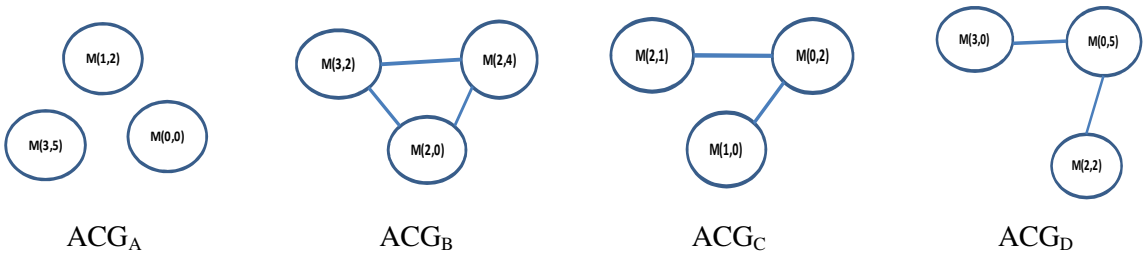


Figure4. 14 *Graphe de conflit d'adresse obtenu après quelques itérations*

À cette étape, l'algorithme sélectionne la colonne la plus contrainte, correspondant à la semi-colonne d'écriture au cycle 2 dans la matrice d'assignation M. Le graphe biparti *G=(D,B,E)* associé à cette colonne est illustré dans la Figure4. 15 . Ainsi à chaque solution d'assignation possible est associée une métrique permettant de sélectionner l'une de ces solutions.

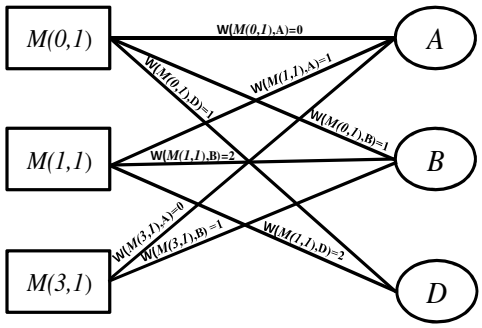


Figure4. 15 *Graphe biparti associée à l'exemple pédagogique*

Puisque le sous graphe ACG_B contient le plus grand nombre de conflits. L'algorithme commence par assigner le banc *B*. L'algorithme affecte ainsi l'arête de poids le plus faible, dans notre cas il s'agit de l'arc *(M(3,1), B)* ou de l'arc reliant *(M(0,1), B)*. Admettons que l'algorithme sélectionne l'arc *(M(3,1), B)*. Ensuite, il met à jour le graphe biparti en supprimant toutes les arêtes violant les contraintes de réseaux et de mémoires (cf. Figure4. 17).

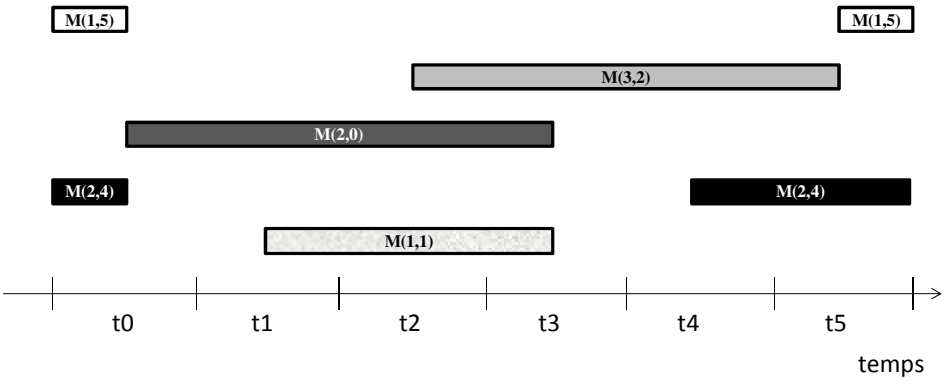


Figure4. 16 *Durées de vie des données assignées au banc mémoire B*

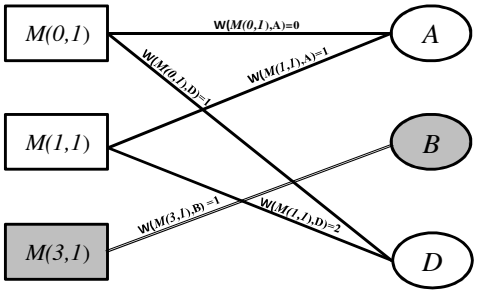


Figure4. 17 *Affectation du banc B à la donnée M(3,1)*

Le résultat de l'heuristique appliquée à l'ensemble de données non assignés de la semi-colonne d'écriture au cycle 2 de la matrice d'assignation *M* est visualisé dans la Figure4. 18.

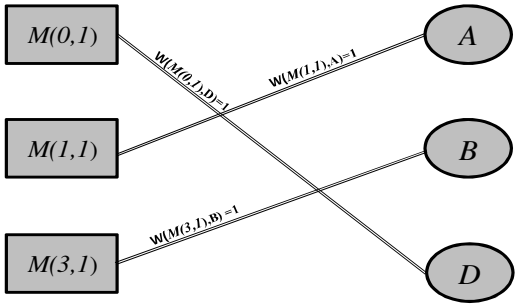


Figure4. 18 *Ensemble d'arêtes pondérées*

Cette heuristique est appliquée à toutes les semi-colonnes non explorées dans la matrice M. Ensuite dans la phase de résolution des conflits, chaque semi-colonne non assignée dans M est assignée à un banc mémoire.

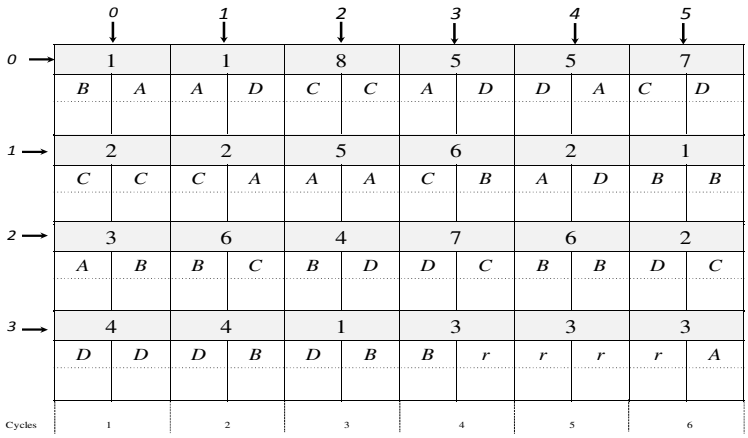
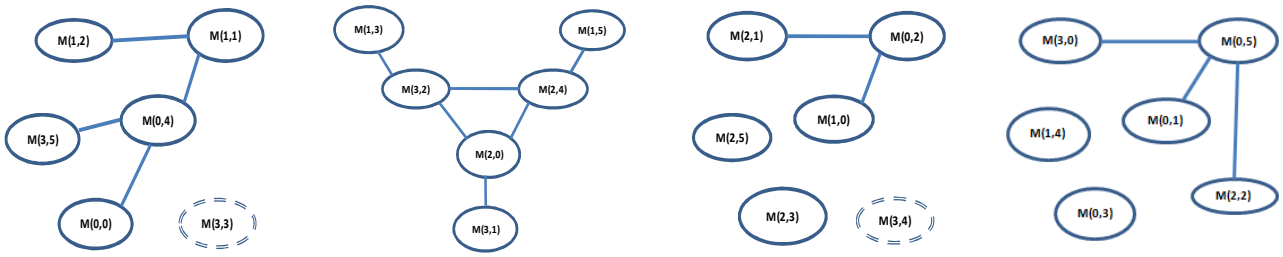


Figure4. 19 *Résultat final de l'assignation mémoire*

Nous obtenons au final la matrice d'assignation mémoire illustrée dans la Figure4. 19 . Dans les graphes ACG correspondant on retrouve par exemple ACG_C : 5 accès aux données sont réalisés dans ce banc - *M(0,2), M(1,0), M(2,1), M(2,5), M(2,3)* - avec deux conflits existant entre *M(0,2), M(1,0)* et entre *M(0,2), M(2,1).* De plus, les accès aux données *M(3,4)* et *M(3,3)* ne sont en relation avec aucun autre nœud des ACG car ils sont mémorisés dans un registre *r*.



(a)Graphe associé au banc A ACG_A

(b)Graphe associé au banc B ACG_B

(c)Graphe associé au banc C ACG_C

(d)Graphe associé au banc D ACG_D

Figure4. 20 *Graphe de conflit d'adresse généré*

L'algorithme commence par sélectionner le nœud le plus contraint. Puisque initialement aucun nœud n'est assigné à une adresse mémoire, le nœud le plus contraint est celui de degrés le plus élevé (connecté aux plus grand nombre d'arcs de conflits). Dans le cas de notre exemple, *M(0,4) M(2,0), M(0,5)* ont le même degré (égale à 3). L'algorithme choisit l'un de ces nœuds : soit *M(0,4)* la première donnée traitée dans le ACG. Il génère ensuite l'ensemble de solutions d'adresses valides pour *M(0,4)*: $S_{M(0,4)}=\{@_A0, @_A1, @_A2, @_A3, @_A4, @_A5\}$, puis sélectionne la première adresse $@_A0$ (Adresse *0* du banc mémoire *A*) dans $S_{M(0,4)}$, et l'assigne à la donnée *M(0,4)* dans son accès en écriture et également dans son accès en lecture (voir Figure4. 21).

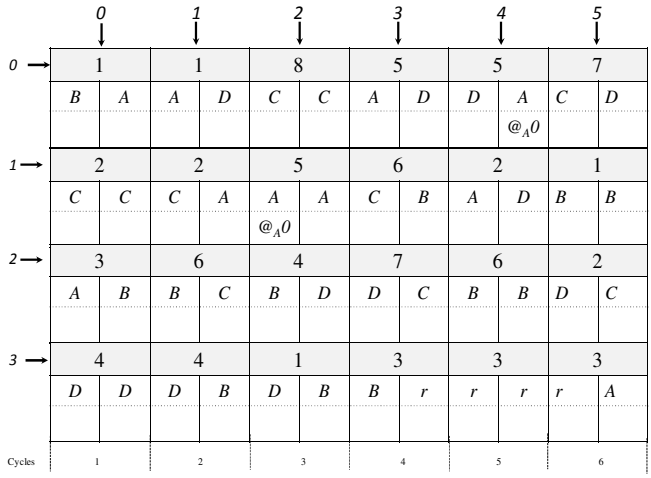


Figure4. 21 *Premier adressage mémoire*

Puis, l'algorithme met à jour les degrés de saturation des différents nœuds du ACG, ainsi M*(0,4)*, M*(3,5)* et M*(1,1)* ont un degré de saturation qui est égale à 1 puisque ces nœuds sont connectés au nœud M*(0,4)* (déjà assigné à l'adresse $@_A0$). Le reste des nœuds du ACG ont un degré de saturation nul. Dans ce cas, c'est le nœud M*(1,1)* qui est sélectionné puisqu'il est connecté à deux arcs de conflits.

L'algorithme d'adressage détaillé dans la Figure4. 10 est ainsi appliqué à l'ensemble des ACG.

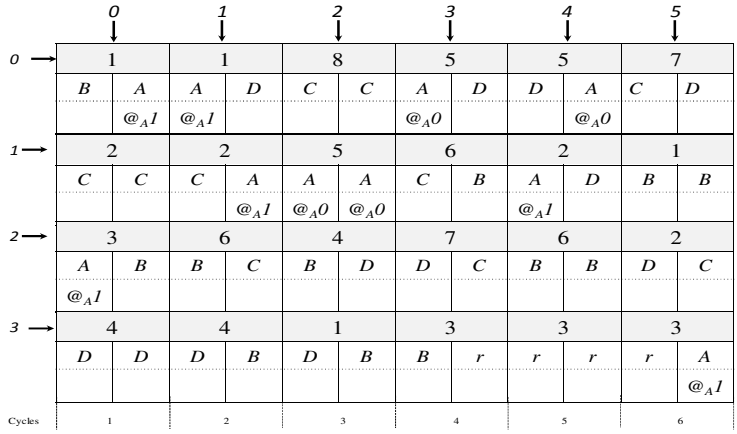


Figure4. 22 *Matrice d'adressage partiellement assignées*

Supposons qu'après quelques itérations l'algorithme sélectionne l'accès *M(3,2)* dans le sous graphe ACG_B (cf. Figure4. 20). Afin d'explorer l'espace des solutions d'adressages, nous utilisons la métrique (4. 4) que nous venons déjà de détailler précédemment..

$$W(n_i,@_jk) = NbrOcc\ (@_fk, T\_accès\ (n_i)) - MoyOccConf\ (@_fk, T\_accès\_Conf\ (n_i)) \qquad 4.\ 4$$

Pour calculer le coût associé à l'affectation de la donnée *M(3,2)* à l'adresse $@_B0$. Cette métrique est décomposée en deux parties : une partie exprimant le nombre d'occurrence de l'adresse *0* dans la semi-colonne d'écriture à la donnée *M(3,2)* (cycle 3), ainsi que le nombre d'occurrence de cette même adresse dans la semi-colonne de lecture de *M(3,2)* correspondante au cycle 6, (cf. Figure4. 22):

NbreOcc (@0) = 1+0.

La seconde partie de la métrique calcule le nombre d'occurrence de *l'adresse 0* dans les semi-colonnes d'accès en lecture et en écriture aux données en conflit d'accès mémoire avec *M(3,2)* dans la matrice d'assignation et d'adressage et qui ne sont pas encore assignées à des adresses mémoires *: M(1,3), M(2,0), M(2,4).* Ce nombre d'occurrence est lui-même divisé par le nombre total des accès de ces données.

Dans notre cas, puisqu'à chacune de ces données conflictuelles correspond deux accès (i.e. deux semi-colonne, une en écriture et une autre en lecture), alors le nombre total de ces accès est égale à *3*2=6.*En effet, cette partie représente la régularité potentielle ainsi perdu si on assigne l'adresse $@_B0$ à la donnée *M(3,2).*

NbreOccConf (M(3,2), @0) = (0+0+1+0+1+0)/6 = 2/3

Par conséquent, le coût asocié à cette solution d'adressage sera *:*

$W(M(3,2),@_B0) = (1+0)-(0+0+1+0+1+0)/6 =1-2/3 = 1/3$

Selon les mêmes calculs, nous obtenons pour toutes les autres solutions d'adressages (adresse 1 ou 2) possibles les valeurs suivantes :

$W(M(3,2),@_B1) = -1/2$

$W(M(3,2),@_B2) = W(M(3,2),@_B3) = W(M(3,2),@_B4) = 0$

En conclusion, la valeur la plus élevée est $W(M(3,2),@_B0) = 1/3$.

Par conséquent l'accès en écriture et en lecture de la donnée *M(3,2)* est effectué à l'adresse 0 dans le banc mémoire *B*. Le même processus est appliqué aux restes des données de la matrice et le résultat final de l'adressage est illustré dans la Figure4. 23 .

	0		1		2		3		4		5	
0	1		1		8		5		5		7	
	B	A	A	D	C	C	A	D	D	A	C	D
	$@_B0$	$@_A1$	$@_A1$	$@_D1$	$@_C0$	$@_C0$	$@_A0$	$@_D1$	$@_D1$	$@_A0$	$@_C1$	$@_D0$
1	2		2		5		6		2		1	
	C	C	C	A	A	A	C	B	A	D	B	B
	$@_C1$	$@_C1$	$@_C1$	$@_A1$	$@_A0$	$@_A0$	$@_C1$	$@_B1$	$@_A1$	$@_D1$	$@_B0$	$@_B0$
2	3		6		4		7		6		2	
	A	B	B	C	B	D	D	C	B	B	D	C
	$@_A1$	$@_B2$	$@_B1$	$@_C1$	$@_B0$	$@_D1$	$@_D0$	$@_C1$	$@_B1$	$@_B1$	$@_D1$	$@_C1$
3	4		4		1		3		3		3	
	D	D	D	B	D	B	B	r	r	r	r	A
	$@_D1$	$@_D1$	$@_D1$	$@_B0$	$@_D1$	$@_B0$	$@_B2$	-	-	-	-	$@_A1$
Cycles	1		2		3		4		5		6	

Figure4. 23 *Matrice d'adressage finale*

L'adressage mémoire générée (cf. Figure4. 22) par l'heuristique que nous proposons est potentiellement régulier, en effet nous avons en moyenne 3 adresses identiques générées parmis 4 à chaque cycle d'horloge (soit en écriture ou en lecture). Ces séquences d'adressages seront ensuite stockées dans des ROMs dédiées. Afin d'exploiter cette régularité nous proposons ainsi une évolution permettant d'optimiser la taille ainsi que le nombre de ces ROMs d'adressage utilisées.

On notera également que cette étape de génération des adresses mémoires pourrait également être utilisée avec d'autres algorithmes de placement mémoire [CHA10a] ou [TAR04] par exemple, mais

sans avoir la possibilité de guider l'assignation mémoire pour l'obtention d'un ACG favorisant la régularité de l'adressage.

III. Compaction des ROMs d'adressages optimisées

A. Transformation des adresses mémoires en matrice de banc et placement des adresses mémoires dans des ROMs dédiées

Pour simplifier nos explications concernant l'optimisation de l'architecture de contrôle à base de ROM, nous réorganisons les séquences d'adressage par banc mémoire et non plus par processeur (cf. Figure4. 24). Par exemple la première ligne correspond aux séquences d'adressage du banc mémoire *A*.

Nota :
Lorsqu'un banc mémoire n'est pas accédé (i.e. un processeur va accéder à un registre à la place) alors au cycle correspondant aucune adresse n'est fournie à ce banc mémoire (e.g. accès en écriture au banc mémoire *A* au cycle 3.). Ainsi dans les séquences d'adressage que nous construisons (cf. figure 4.25) feront apparaître une « case vide ».

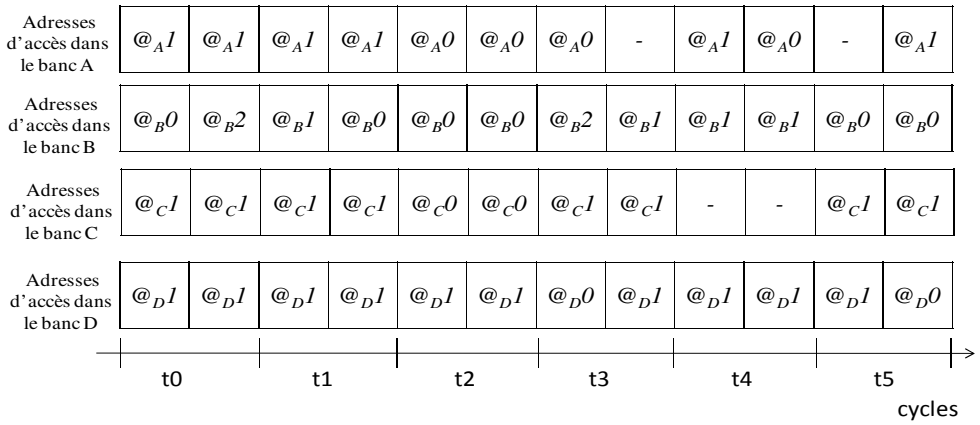


Figure4. 24 *Séquences d'adressage aux bancs mémoire*s

Afin de contrôler les différents bancs mémoire, ces séquences d'adressages sont stockées dans des ROMs ; par exemple la ROM contrôlant le banc mémoire A mémorise à son adresse *0* l'adresse *1* d'accès à la $RAM_A(@_A1)$ (cf. Figure4. 25). La présence de cases vides dans la semi-colonne d'écriture du cycle 3 et la semi-colonne de lecture du cycle 5 montrent que durant ces périodes la RAM_A n'est accédée par aucun processeur. Nous verrons ultérieurement que ces cases vides seront exploitées pour favoriser la compaction de ces ROMs d'adressage.

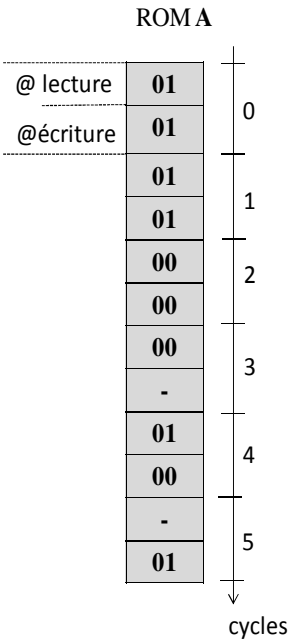


Figure4. 25 *ROM d'adressage pour le contrôle du banc mémoire A*

Nous avons choisit de stocker dans ces ROMs d'adressages uniquement les adresses mémoires sans leur associer les signaux de contrôles (write enable). En effet, puisque dans notre approche certains accès mémoires sont déportés des bancs mémoires vers les registres additionnels, les signaux d'écriture des bancs mémoire (RAMs) correspondants ne seront dans ce cas pas activé : le signal « *write enable* » de ces RAMs sera forcé à '0' dans certains cycles d'écritures (à priori un cycle sur deux) quand ces RAMs ne sont accédées par aucun processeur. Notre objectif est ainsi de favoriser la régularité des séquences d'adressage mémorisées dans les ROMs. Il s'agit là d'une première solution que nous souhaitons évaluer, mais l'algorithme de compaction et de fusion que nous proposons est capable à partir d'un niveau de régularité même minimal d'optimiser la taille des ROMs.

B. Compaction et Fusion des ROMs

Suite à l'analyse que nous venons de présenter, nous savons que 4 ROMs sont nécessaires, à priori, pour contrôler les accès aux mémoires RAMs de l'architecture. Pour ce faire, nous proposons un flot dédié à la réduction de la taille et du nombre de ces ROMs (cf. Figure4. 26). Ce flot s'insère à la suite du flot proposé section II.

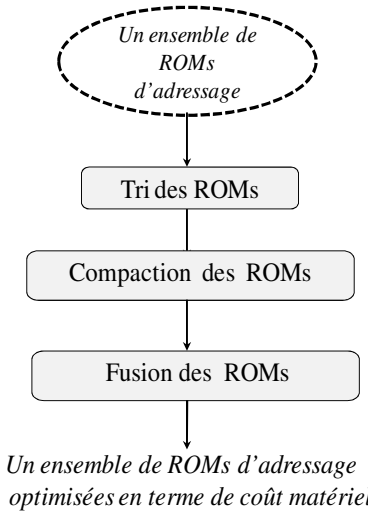


Figure4. 26 *Flot de conception de la Compaction&Fusion des ROMs*

a) Tri des ROMs

Nous définissons tout d'abord des terminologies qui seront utilisées pour trier les ROMs d'adressage.

Définition Degré de Compatibilité entre deux ROMs

Le degré de compatibilité de deux ROMs i et j est evalué de la manière suivante :,

$$\text{Degré de Compatibilité } (\text{ROM}_\text{i}, \text{ROM}_\text{j}) = \sum_{\text{t}\in [0,\text{T}-1]} R_{ij}\ (\text{t}, \text{ROM}_\text{i}, \text{ROM}_\text{j})$$

Avec,

T représentant le nombre total de cycles de traitements durant lesquelles les processeurs traitent les données.

R_{ij} est une métrique dont la valeur dépend des adresses stockées dans ROM_i et ROM_j. Plus concrètement, cette métrique exprime la compatibilité des adresses stockées dans ROM_i avec les adresses stockées dans ROM_j et qui sont lues respectivement par RAM_i et RAM_j dans le même cycle de traitement *t*.

Nota :

Nous supposons que durant un cycle de calcul t, les processeurs effectuent en parallèles deux accès : le premier est pendant [0 .. t/2] pour lire une valeur et le deuxième est pendant [t/2 .. t] pour écrire un résultat.

Définition Degré de compatibilité entre une ROM et un ensemble de ROMs

DegréCompatibilité (ROM_i, sROM)=$\sum_{k=0;k\neq i}^{k=p} Compatibilité\ (ROM_i, ROM_k)$

Avec sROM = {ROM_0,….ROM_{p-1}}.

Entrées :

Deux ROMs: ROM_i et ROM_j contrôlant respectivement deux RAMs: RAM_i et RAM_j.
Le cycle de traitement *t*.

Sorties :

Valeur de R_{ij} : La compatibilité des adresses stockées respectivement dans ROM_i et ROM_j et lus par RAM_i et RAM_j dans le cycle *t*.

Début

Si RAM_i n'est accédée en lecture par aucun processeur dans le cycle *t* (présence d'un registre) **alors**
R_{ij}[0 .. t/2]=1
Sinon
Si RAM_i et RAM_j sont accédées en lecture dans la même adresse mémoire **alors**
R_{ij}[0 .. t/2]=1
Sinon
R_{ij}[0 .. t/2]=0
Fin Si
Fin Si
Si RAM_i ou RAM_j n'est accédée en écriture par aucun processeur dans le cycle *t* (présence d'un registre) **alors**
R_{ij}[t/2 .. t] =1
Sinon
Si RAM_i et RAM_j sont accédées en écriture dans la même adresse mémoire **alors**
R_{ij}[t/2 .. t] =1
Sinon
R_{ij}[t/2 .. t] =0
Fin Si
Fin Si

Fin

Figure4. 27 *Compatibilité des adresses mémoires envoyées par deux ROMs dans un cycle de traitement particulier*

A partir de ces définitions, la prochaine étape va donc consister à trier les ROMs selon leurs degrés de compatibilité entre elles (leur similitude les unes par rapport aux autres). Nous considérons comme *ROM principale* celle qui a une compatibilité maximale avec le reste des ROMs.

Entrées :

Un ensemble de ROMs $sROM=\{ROM_0,\ldots\ldots ROM_{p-1}\}$ non trié.

Sorties :

sROM trié.

Début

Pour toutes $ROM_i \in sROM$
Calculer le degré de compatibilité de ROM_i avec le reste des ROMs dans *sROM*
Fin Pour
Trier les différentes ROMs suivant l'ordre décroissant de leurs degrés de compatibilité avec *sROMs*.

Fin

Figure4. 28 *Tri des ROMs*

b) Compaction des ROMs

L'identification de la *ROM* principale va nous permettre d'éliminer toutes les séquences d'adresses identiques, ordonnancées au même cycle (i.e. à la même adresse dans la ROM), répétées entre les ROMs. Au final, pour chaque séquence d'adressage répétée dans plusieurs ROMs aux mêmes cycles, il n'y aura plus qu'une seule ROM la mémorisant.

L'opération de tri expliqué précédemment permet de définir une certaine priorité dans ce classement, dans le sens où si une même séquence d'adressage est stockée dans deux ROMs et envoyée dans le même cycle d'horloge, alors c'est la ROM de rang inférieur qui mémorisera la séquence de contrôle. C'est pourquoi la ROM caractérisée par le degré de compatibilité le plus élevée par rapport aux autres est considérée comme la ROM principale : c'est elle qui pilotera le plus grand nombre de RAMs. Par conséquent, cette ROM ne subira aucune modification à ce stade de l'algorithme de compaction.

L'optimisation sera ainsi appliquée aux ROMs secondaires en se basant sur le principe suivant: Si un ensemble de ROMs envoient la même séquence de contrôle aux RAMs qui leur sont associées, dans le même cycle d'horloge, alors c'est la ROM possèdant le degré de compatibilité le plus élevé qui stockera la séquence de contrôle, et c'est elle qui contrôlera les différentes RAMs concernées. Par voie de conséquence, les séquences de contrôle en question seront supprimées des autres ROMs, réduisant potentiellement leur taille. La *Figure4. 29* détaille l'algorithme d'élimination des séquences de contrôle répétées.

Entrées :

Un ensemble de ROMs trié: sROM={ROM_0,……..ROM_{p-1}} avec rang(ROM_i)<rang(ROM_j) si i<j

Sorties :

sROM optimisé en termes de taille

Début

Pour tout cycle *t* € *[0, T-1]*
Pour tous *i* allants de 1 jusqu'à *p-1*
Pour tout *j* allant de 0 jusqu'à *i-1*
Si il existe une adresse de lecture envoyée par ROM_i dans le cycle *t* **alors**
Si l'adresse de lecture envoyée par ROM_i est la même que l'adresse de lecture envoyée par ROM_j **alors**
Supprimer l'adresse stockée dans ROM_i
Fin Si
Fin Si
Si il existe une adresse d'écriture envoyée par ROM_i dans le cycle *t*.
Si l'adresse d'écriture envoyée par ROM_i est la même adresse d'écriture envoyée par ROM_j **alors**
Supprimer l'adresse stockée dans ROM_i
Fin Si
Fin Si
Fin Pour
Fin Pour
Fin Pour

Fin

Figure4. 29 *Algorithme de compaction des ROMs*

Nous appliquons ainsi l'algorithme de tri et de suppression de séquences contrôle répétées sur notre exemple pédagogique. Le calcul des degrés de compatibilités des différentes ROM donnent les resultats suivants :

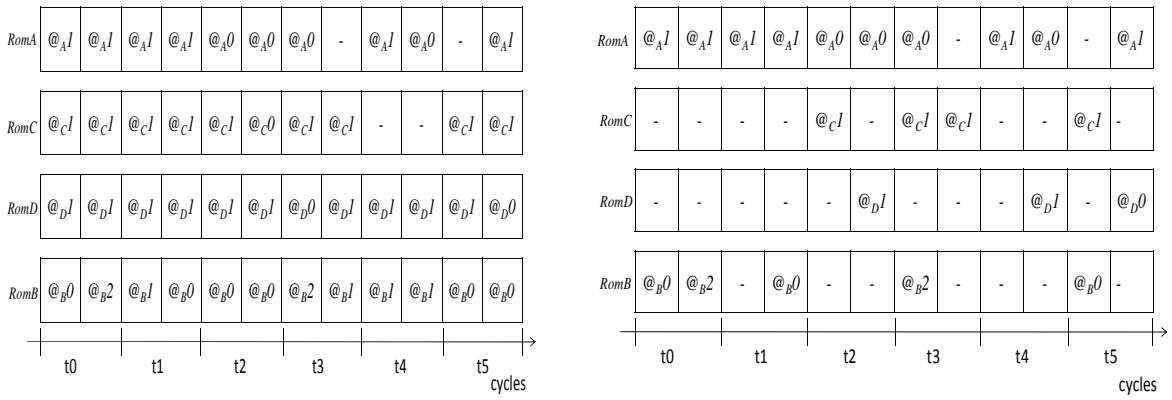
Soit sROM={ROM_A, ROM_B, ROM_C, ROM_D},

DegréCompatibilité (ROM_A, sROM)=22
DegréCompatibilité (ROM_B, sROM)=12
DegréCompatibilité (ROM_C, sROM)=22
DegréCompatibilité (ROM_D, sROM)=17

On obtient,

sROM_trié= {ROM_A, ROM_C, ROM_D,ROM_B}

Dans ce cas de figure, ROM_A est la ROM principale.



(a) ROMs avant optimisation (b) ROMs après optimisation

Figure4. 30 *Application de la compactions des ROMs sur l'exemple pédagogique*

La Figure4. 30.b illustre le résultat de la compaction des ROMs. On peut ainsi observer que la ROM_A est considérée comme la *ROM principale*. C'est cette ROM qui ici stock le plus grand nombre de séquences d'adressage. Plusieurs séquences d'adressage ne sont donc plus nécessaires dans les autres ROMs. Ainsi, grâce à la maximisation de la régularité des séquences d'adressage on peut minimiser la taille des ROMs.

c) Fusion des ROMs

L'élimination des séquences de contrôles répétées crée des « cases vides » dans les différentes ROMs. Ces cases mémoires peuvent elles-même être utilisées pour compacter encore l'ensemble des ROMs de contrôle. À titre d'exemple, une ROM peut être éliminée si il est possible de déplacer toutes les adresses mémoires quelle stocke vers des ROMs de rangs inférieurs. Par exemple, l'ensemble des adresses contenues dans ROM_D peut être regroupé dans le banc ROM_C.

L'objectif par cette étape est de garder au finale un minimum de ROMs pour contrôler toutes les RAMs. Pour ce faire, nous proposons un algorithme dédié à la fusion des ROMs (cf. *Figure4. 31*). Celui ci commence par déplacer les adresses de la ROM classé deuxième vers la ROM principale, ensuite il traite à chaque fois la ROM de rang juste supérieur et commence en priorité par vérifier s'il est possible de déplacer ses adresses vers la ROM principale si ce n'est pas le cas il vérifie la possibilité dans les ROMs secondaires et ainsi de suite jusqu'à vérifier la possibilité de faire un déplacement des adresses mémoires vers toutes les ROMs de rangs inférieurs (cet algorithme s'inspire d'un Left-Edge [Mur00]).

Entrées :

Un ensemble de ROMs triée et compactée $sROM = \{ROM_0, ... ROM_{p-1}\}$, avec $rang(ROM_i) < rang(ROM_j)$ si i<j

Sorties :

Un ensemble de ROMs fusionnée

Début

Pour tout *i* allant de 1 jusqu'à *p-1*
Pour tout *k* allant de 0 jusqu'à *i-1*
Pour tout $t \in [0, T-1]$
Si aucune adresse de lecture n'est lu à partir de la ROM_k et il existe une adresse de lecture $@i_t$ lu à partir de la ROM_i dans le cycle *t* **alors**
Déplacer l'adresse de lecture $@i_t$ de la ROM_i vers la ROM_k
Fin Si
Si aucune adresse d'écriture n'est lu à partir de la ROM_k et il existe une adresse d'écriture $@i_t$ lu à partir de la ROM_i dans le cycle *t* **alors**
Déplacer l'adresse d'écriture $@i_t$ de la ROM_i vers la ROM_k.
Fin Si
Fin Pour
Fin Pour
Fin Pour

Fin

Figure4. 31 *Algorithme de fusion ROMs*

La Figure4. 32 illustre les différentes étapes de l'algorithme de fusion des ROMs appliquées à notre exemple pédagogique. Dans (2) deux séquences d'adresses initialement stockées dans la ROM_C ont éte déplacées vers la ROM principale grâce à la présence dans celle-ci de deux cases vides. Dans le (3) trois séquences d'adresses ont été déplacées de la ROM_D vers la ROM_C puisque la ROM principale est complètement remplie. Finalement dans (4),après avoirs libéré plusieurs cases dans ROM_D et ROM_C, toutes les séquences d'adresses mémorisées dans la ROM_B ont été déplacées vers la ROM_C et la ROM_D.

Nous avons au final compacté les 4 ROMs initales en 3 ROMs dont une principale de taille maximale et deux autres secondaires de tailles inférieurs.

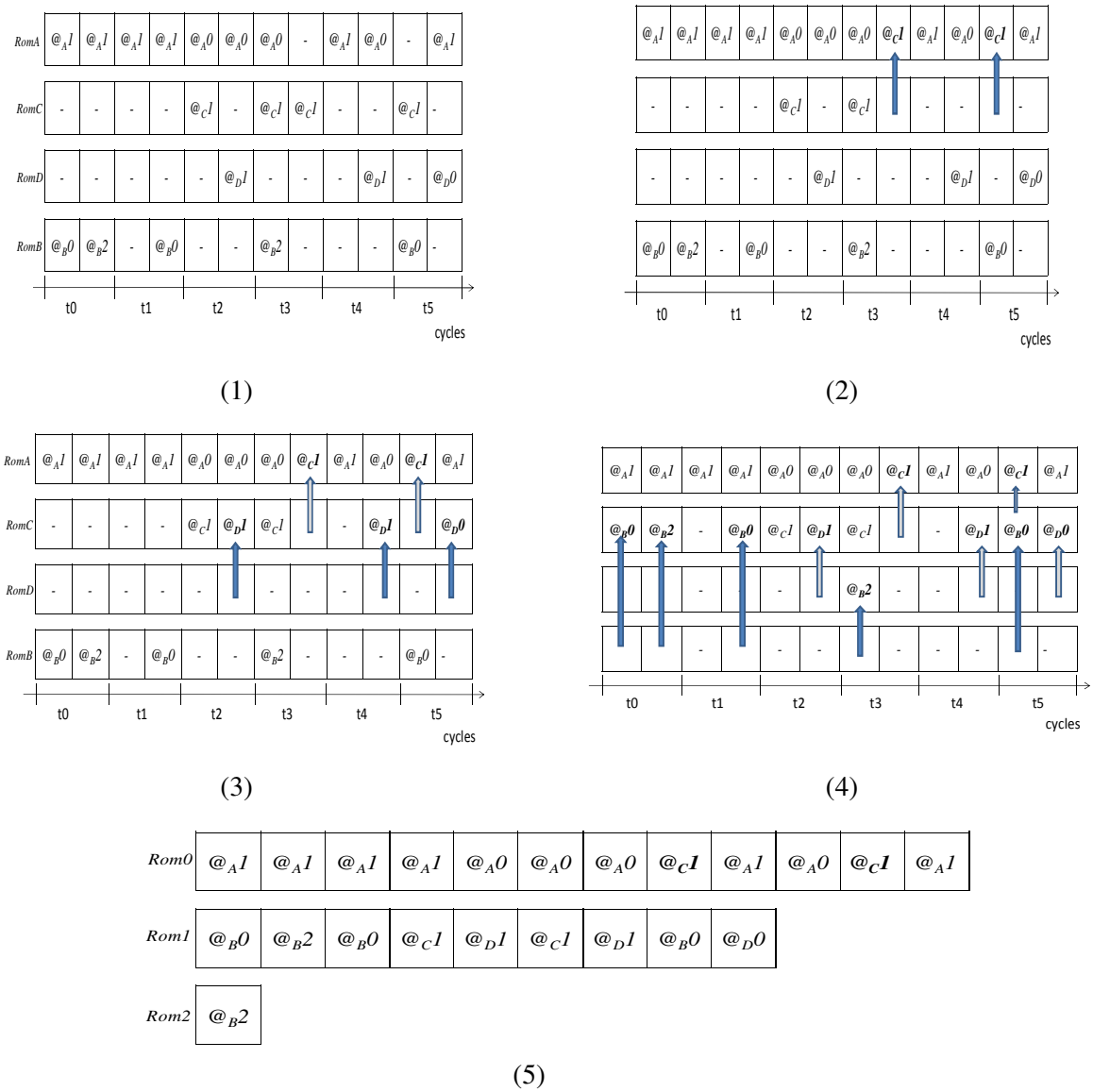


Figure4. 32 *Application de l'algorithme de fusion des ROMs sur l'exemple pédagogique*

C. Architecture RTL schématique

La Figure4. 33 illustre l'architecture RTL schématique générée dans le cadre de notre exemple pédagogique. Elle se compose de 4 bancs mémoires (Mem_0, Mem_1, Mem_2, Mem_3), 4 processeurs (PE_0, PE_1, PE_2, PE_3) et de deux réseaux papillons pour la communication entre les processeurs et les bancs mémoires (accès en écriture et en lecture). Un registre R_0 est également présent permettant de mémoriser transitoirement les données conflictuelles.

L'unité de contrôle est divisée en trois unités, une contrôlant la mémoire, une autre est dédiée au contrôle du registre et finalement une qui pilote les deux réseaux. Le contrôleur mémoire est composé d'une ROM principale qui pilote toutes les mémoires et de deux ROMs secondaires de tailles plus petites pilotant chacune un sous-ensemble de mémoires. Chaque ROM est piloté par son propre compteur qui s'incrémente si celle ci est utilisée pour piloter au moins une mémoire. Un automate (FSM, pour Finite State Machine) est ainsi nécessaire pour piloter la lecture des ROMs. Les signaux de contrôles pilotant les deux réseaux sont stockés dans une ROM dédiée. Enfin, un second automate se charge du contrôle du registre et du multiplexeur associé.

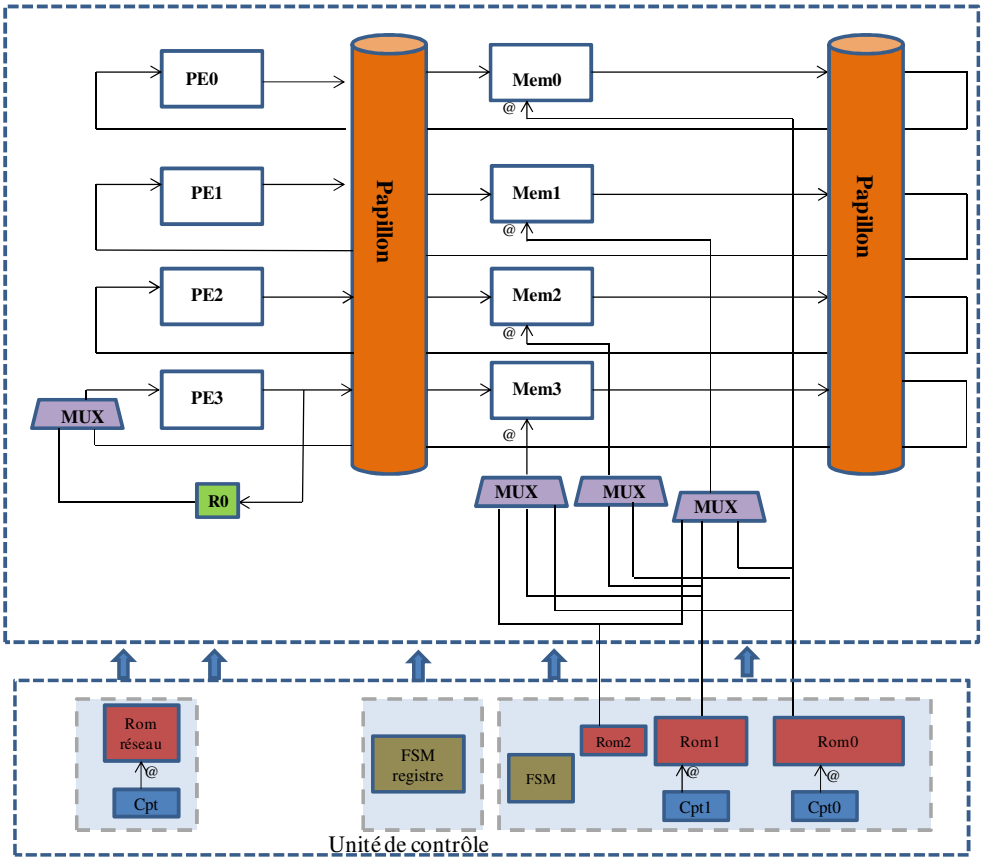


Figure4. 33 *Architecture RTL générée*

L'approche que nous proposons permet d'automatiser l'adressage des données dans leur banc mémoire tout en garantissant des accès aux bancs mémoires sans conflit et en respectant un réseau d'interconnexion cible. L'objectif de cette approche est de maximiser la régularité au niveau de des séquences d'adressage en se basant sur une fonction de coût.

Toutefois, et nous l'avons évoqué dans le chapitre précédent, pour certaines applications ayant une loi de permutation totalement incompatible avec le réseau d'interconnexion ciblé par le concepteur, le coût lié à l'utilisation de registres additionnels peut devenir trés important. De faite, il est théoriquement possible que dans certains cas d'application ce surcoût limite fortement les gains apportés par notre approche d'optimisation du contrôleur.

C'est pour répondre à ce défi qu'une collaboration a été initiée avec Saeed Ur Reehman, un doctorant du laboratoire. Nous proposons cette fois un flot de conception généralisée permettant la génération automatique d'un entrelaceur mémoire parallèle sans conflit optimisé (coût d'implémentation matérielle) capable de s'affranchir de cette nouvelle problématique.

IV. Bilan

Dans ce chapitre nous avons présenté le flot de conception permettant la génération automatique des entrelaceurs mémoires parallèles sans conflits, sous contrainte de réseau et optimisés en terme de coût du contrôleur de l'architecture. Nous avons proposé également une stratégie permettant d'optimiser l'architecture du contrôleur via compaction et fusion des ROMs. Des exemples à visée pédagogique ont été présentés afin d'expliquer l'implémentation des modèles que nous proposons.

Flot de conception généralisé pour la génération automatique des entrelaceurs mémoires parallèles

Ainsi que nous l'avons déjà évoqué, l'approche de placement mémoire basée sur la relaxation de la contrainte mémoire, combinée à l'optimisation à la volée du coût du contrôleur permet d'obtenir des gains significatifs en terme de coût architectural. Toutefois, si le concepteur souhaite obtenir une architecture basée sur un réseau d'interconnexion trop éloigné de ce que peut supporter nativement la règle d'entrelacement ciblée, le surcoût apporté par l'ajout de registres (et la logique d'aiguillage associée) peut dans certain cas faire perdre tout intérêt à l'approche.

Ainsi, dans le cadre d'une collaboration menée avec Saeed Ur Reehman, nous avons proposé l'extension du principe de relaxation de contrainte au réseau d'interconnexion. Dans ce cas précis, là où la relaxation mémoire ajoute des registres à l'espace mémoire disponible pour mémoriser les données, la relaxation du réseau va ajouter des permutations élémentaires dans le réseau d'interconnexion pour offrir un ensemble de permutations disponibles plus élevé. L'idée en combinant ces deux axes d'exploration, est de chercher à déterminer le meilleur compromis architectural possible entre d'une part les désidératas du concepteur, mais également la règle d'entrelacement et le coût de l'architecture. La Figure4. 34 présente un d'un flot d'exploration généralisé permettant l'exploration de l'espace des solutions architecturales combinant les deux approches de relaxation.

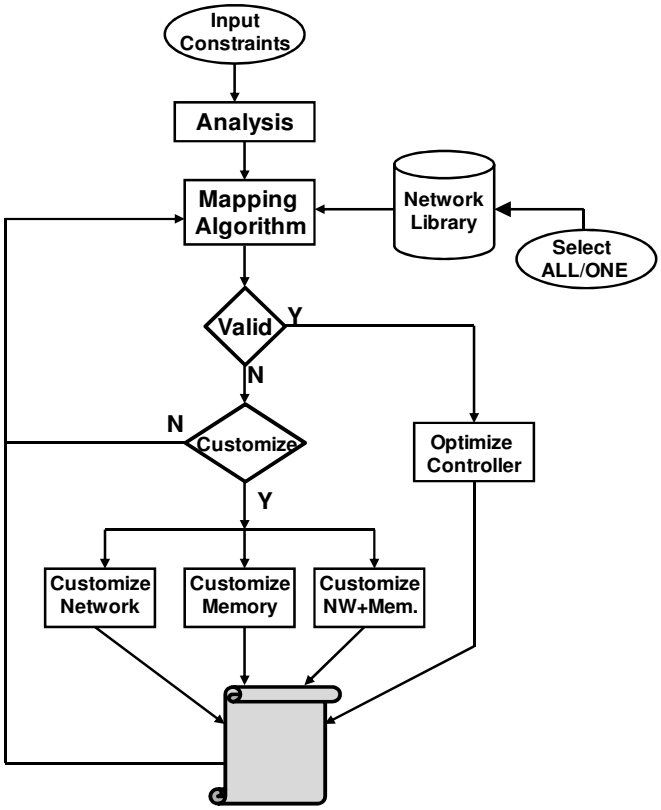


Figure4. 34 *Flot de conception générale*

Le principe est le suivant : le concepteur fournis un ensemble de contraintes (parallélisme, type de réseau cible, règle d'entrelacement) qui seront ensuite exploités par les algorithmes de placement mémoire. Un ensemble de réseau d'interconnexion possible est également fourni en entrée. Le principe est de laisser la liberté à l'utilisateur de déterminer s'il souhaite voir notre approche lui fournir un ensemble de solutions de placement mémoire possibles (en fonction des différents réseaux

utilisables depuis la librairie), ou bien s'il est près à payer un éventuel surcoût pour avoir la certitude d'obtenir une architecture exploitant de façon certaine le réseau d'interconnexion qu'il a sélectionné.

A partir de ces informations, une première étape d'analyse va déterminer si compte tenu de la règle d'entrelacement et du parallélisme de traitement de l'architecture, il sera possible d'utiliser un placement mémoire de type « in place » (i.e. la donnée est systématiquement mémorisée dans le même banc mémoire et à la même adresse) ou si notre approche de double placement mémoire devra être utilisée. Ceci permettra accessoirement d'optimiser l'exploration de l'espace des solutions, mais surtout cela impactera fortement l'architecture du contrôleur mémoire qui sera généré comme nous l'avons vu précédemment.

Ensuite, un placement mémoire sans conflit est réalisé par notre approche, en désactivant toutefois la relaxation de la mémoire afin de déterminer si la règle d'entrelacement peut respecter ou non le réseau cible initialement définit par le concepteur. Si c'est le cas l'architecture correspondante est générée. Sinon, ou bien si le concepteur autorise notre flot à explorer et proposer d'autres solutions architecturales (étape dite de « personnalisations »), notre approche va générer des placements de données sans conflits mémoires en exploitant les différentes approches de relaxation séparément (relaxation réseau, relaxation mémoire) ou en les combinant (relaxation réseau et mémoire). Un ensemble de solution est ainsi généré.

Le flot proposé n'est à ce jour pas complètement réalisé, en particulier l'optimisation du contrôleur mémoire proposée dans cette thèse n'est pas encore intégré, et la génération du code VHDL final n'est pas fonctionnelle. Les résultats que nous présentons dans ce manuscrit sont donc issus d'estimations exploitant les mêmes approches méthodologique et technologique que dans [SAN12]. De plus, ces travaux faisant l'objet d'un article en cours de soumission, seuls quelques résultats expérimentaux représentatifs seront donnés dans le chapitre suivant.

Un ensemble d'expérimentations ont étés menées afin de valider sur des standards de communication les approches que nous proposons. Nous présentons ces travaux dans le chapitre suivant.

Chapitre 5

Expérimentations

Ce chapitre présente une série d'expériences réalisées sur des applications du domaine des Télécommunications. Les premières expérimentations montrent la pertinence des solutions adoptées pour les algorithmes de placement de données en mémoire sans conflit via relaxation de la contrainte mémoire. Ensuite, nous présentons plusieurs cas d'études visant à montrer le potentiel de notre approche complète, c'est-à-dire en intégrant l'optimisation du contrôleur. Enfin, nous présentons les premiers résultats de nos travaux dans le cadre de la mise au point d'un flot d'exploration généralisé.

I. Introduction

Dans ce chapitre nous exposons un ensemble d'expérimentations visant à démontrer la pertinence des approches théoriques et algorithmiques que nous avons proposées. Nous exposerons ainsi des résultats de synthèse d'entrelaceurs réalisés à l'aide de flots de conception mettant en œuvre les méthodes proposées dans le cadre de ces travaux. Les cas d'études considérés sont issus des derniers standards de communication. L'espace des solutions a été exploré en faisant varier les contraintes de placement mémoire, les réseaux d'interconnexion cibles, les parallélismes et les longueurs de trames. Les résultats de synthèse que nous obtenons sont comparés à ceux obtenus à l'aide d'approches comparables de l'état de l'art.

Dans [SAN12] les auteurs ont montré que pour les différentes configurations architecturales qu'ils ont explorées, la surface des contrôleurs représente l'essentiel du coût de l'architecture comparativement au coût des processeurs SISO ou du réseau (estimations de surface en portes NAND équivalent, PNE, basées sur une technologie 90nm de STMicroelectronics). Tirés de ces séries d'expérimentations, la Figure 5. 1 illustre un cas d'étude basé sur le standard LTE avec un réseau d'interconnexion cible de type réseau de Benes avec 32 processeurs SISO radix 2 (parallélisme de traitement). Les surfaces sont fournies en PNE. Nous constatons dans cet exemple que la surface de la partie contrôle occupe 77% de la surface de l'architecture totale du décodeur. Ceci illustre le fait que les parties contrôles représentent les éléments le plus coûteux d'un point de vue architectural, et constituent ainsi un axe d'optimisation très important. Ainsi, d'une manière générale, nos résultats se concentrent sur les comparaisons des parties les plus pertinentes des architectures générées.

Réseau d'interconnexion (Benes)	Mémoire extrinsèque	Contrôleur	SISO radix2	Total
18.9K	441K	2792K	416K	3668 K

Figure 5. 1 *Surface en portes logiques d'un turbo-décodeur utilisé dans le standard LTE*

La première étude que nous présentons porte sur la mise en œuvre d'un entrelaceur pour le standard HSPA évolution [HSP08], qui présente de nombreux conflits mémoire pour tout degré de parallélisme. Cette première série d'expériences valide les options algorithmiques que nous avons proposées dans le chapitre 3 tout en révélant l'impact de la contrainte de réseau sur le coût de l'architecture finale. Nous montrerons également l'apport de l'optimisation du contrôleur mémoire sur le coût architectural final, en utilisant cette l'approche présentée dans le chapitre 4.

Dans une seconde partie, nous étudierons l'entrelaceur de données d'une application de type UWB. Les expérimentations furent menées en ciblant soit un réseau d'interconnexion fortement contraint (e.g. un barrel-shifter), soit un réseau plus souple (mais plus coûteux) de type cross bar. Les résultats obtenus sont comparés avec ceux obtenus à l'aide d'approches issues de l'état de l'art.

Un troisième ensemble d'expérimentations concerne des applications de type LDPC. Les architectures de décodeurs LDPC aléatoires souffrent, comme nous l'avons vu, de problèmes de conflits mémoire, et d'un coût matériel important. Néanmoins notre approche est capable de générer pour cette classe de codes LDPC une architecture sans conflit mémoire. Les résultats obtenus en termes de coût matériel sont comparés aux approches de l'état de l'art. Le flot de conception que nous proposons est ensuite utilisé pour générer des architectures partiellement parallèles de codes LDPC structurés.

Dans une dernière expérience, nous présenterons succinctement, car les travaux sont toujours en cours, notre flot d'exploration généralisée de placement mémoire. Ces travaux menés en lien avec

Saeed Ur Reehman dans le cadre de ses travaux de thèse exploitent l'approche de placement mémoire et d'optimisation que nous proposons, et s'inspirent de notre principe de relaxation de contraintes pour l'étendre à d'autres paramètres architecturaux (i.e. le réseau d'interconnexion).

Rappel du flot de conception

L'approche que nous avons proposée dans le chapitre 4 peut se résumer sous la forme d'un flot de conception présenté en Figure 5. 2, illustrant les différentes étapes de celui-ci. Ainsi, à partir d'un fichier de contraintes d'entrelacement (i.e. une description de la règle d'entrelacement et une contrainte de réseau), l'outil génère dans un premier temps un placement sans conflit des données, et un graphe de conflit d'adresses *ACG*. Les approches présentées dans la chapitre 4 trouvent également leur place dans ce flot générique, à la différence près que la génération de l'ACG et la fusion des ROMs ne seront pas exploitées.

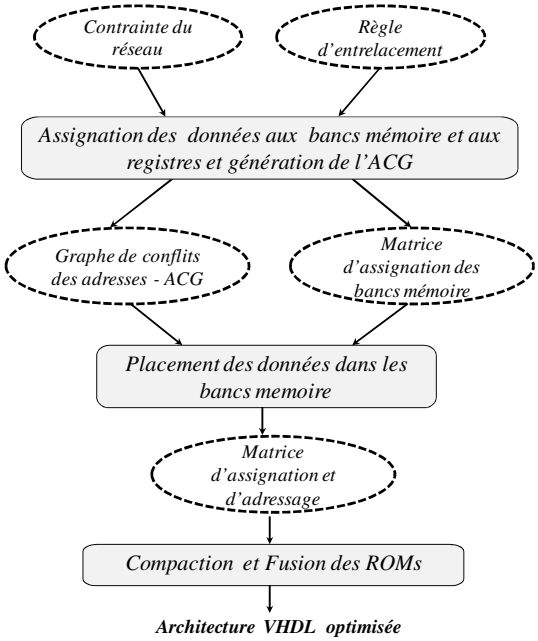


Figure 5. 2 *Flot de conception générique*

Ensuite, à partir de ces deux sources d'informations, l'outil détermine l'adressage des données dans leurs bancs mémoire respectifs. Les séquences d'adressage mémoire étant stockées dans des ROMs dédiées (cf. Chapitre 4), un algorithme d'optimisation de ces ROMs est appliqué pour générer l'ensemble des informations relatives à la génération de l'architecture VHDL (i.e. les tailles des ROMs, les signaux les contrôlant, la logique d'aiguillage…). Finalement, l'outil génère une architecture matérielle optimisée sous la forme d'un fichier VHDL RTL synthétisable.

Rappel :

Pour l'instant si la compaction des ROMs de contrôle (réduire la taille des ROM) est fonctionnelle, l'étape de fusion (réduire le nombre de ROMs) n'est pour l'instant par finalisée, les résultats présentés n'en tirent donc pas bénéfice.

```
number bank = 7
number data = 84

Tab  =

2         3         9         10        13        14        25
2         6         7         8         12        14        15
4         5         6         8         12        15        16
1         3         9         10        16        17        25
3         7         10        11        17        18        25
5         6         8         12        13        18        19
3         4         10        11        19        20        25
2         3         7         10        20        21        25
1         5         6         8         12        21        22
6         8         11        12        22        23        25
3         4         9         10        23        24        25
1         6         8         12        13        24        25
```

Figure 5. 3 *Exemple d'un fichier de contraintes tiré du standard WIMAX*

Les résultats obtenus dans le cadre de ces expériences sont présentés en termes de nombre de slices (Total Slices). Nous avons également mentionné dans nos résultats expérimentaux deux autres types d'informations données par les outils ISE : le nombre de Slices Registre et le nombre de Slices LUT. Notre objectif est de pouvoir observer plus en détails les coûts de différentes parties d'une architecture synthétisée. Les différentes architectures générées ont été synthétisées sur une cible FPGA Xilinx Virtex 6 (ref. xc6vlx75t-3-ff484), en utilisant l'outil de synthèse Xilinx ISE 12.3. Les RAMs (mémoires extrinsèques) sont synthétisées en exploitant les mémoires internes de la carte FPGA tandis que les ROMs générées par nos approches sont synthétisées sous forme de mémoires distribuées (slice LUTs), ce qui permet d'avoir des analyses plus fines concernant la surface du contrôleur mémoire (basée sur des ROMs) puisque celle-ci sera déterminée en fonction du nombre de slices.

Les résultats que nous exposons dans ce manuscrit ne font références qu'à la surface de la partie contrôle (mémoires et réseau), ainsi qu'aux éventuels ajouts de registres impliqués par notre approche de relaxation mémoire. Les résultats que nous proposons ne visent pas à optimiser la taille totale des bancs mémoires utilisés dans l'architecture. Ainsi, puisque les coûts de ces bancs mémoires sont constants, ils n'ont aucun impact différenciant sur les résultats de synthèse. Ils ne présentent donc pas d'intérêt dans le cadre de nos démonstrations et ne seront pas présentés.

Nous verrons au travers de ces expérimentations que le facteur clef pour pouvoir optimiser les architectures d'entrelaceur réside dans un élément dont le coût architectural lui-même est relativement faible : le réseau d'interconnexion. Nous verrons que ceci reste valide pour les LDPC, comme pour les Turbo Codes. Il est ainsi important, une fois le bon réseau d'interconnexion défini par le concepteur, de proposer une approche capable de respecter ce réseau d'interconnexion, mais de le faire intelligemment pour espérer minimiser le coût final de l'architecture. Enfin, déterminer le bon réseau d'interconnexion pouvant s'avérer une tâche ardue pour le concepteur, un flot de conception dédié s'avère nécessaire si l'on souhaite explorer le plus efficacement possible l'espace des solutions.

II. Conception des architectures d'entrelaceurs parallèles pour les Turbo-codes

Les architectures parallèles de Turbo-Code sont souvent complexes en termes de coût d'implémentation matérielle. Pour rappel, l'approche de placement mémoire avec optimisation du contrôleur que nous proposons est capable de trouver une allocation mémoire sans conflit et de cibler également n'importe quel réseau d'interconnexion définie par le concepteur. Ceci est valable pour toute règle d'entrelacement définie dans tout standard de communication. De plus, la prise en compte de l'adressage des données dans les mémoires dès l'exploration de la répartition de ces données dans les dits blocs mémoires, permet d'optimiser le coût du contrôleur mémoire. Dans le cadre d'un

premier ensemble d'expériences, nous implémentons des entrelaceurs utilisés dans deux grands standards actuels : *HSPA Evolution* [HSP08] et l'*Ultra-Wide-Band* [IEE07].

A. Entrelaceur utilisé dans HSPA Evolution

Le système UMTS (3ème génération ou 3G) a été conçu pour améliorer le transport des données (assez peu efficace sur un réseau 2G). Ainsi sous l'égide de 3 GPP *(Third generation patternchip Project*). Le système UMTS a évolué plusieurs fois pour améliorer le débit de données. On distingue ainsi trois principales normes définissant le transfert de données entre l'antenne Relais et le téléphone mobile dans le réseau de 3ème génération : W-CDMA, HSPA, HSPA+ ;

- W-CDMA : connu sous le nom de Release 99, est la première version du système UMTS déployé par les opérateurs à partir de 2001. Son débit maximal pour un transfert descendant (téléchargement) est de 1,920Mb/s.
- HSPA : évolution du W-CDMA ayant pour objectif d'augmenter le débit de données et de diminuer la latence en réorganisant la manière d'effectuer les transferts dans des canaux spécialisés.
- HSPA+ : évolution de la norme HSPA, avec pour objectif d'améliorer encore une fois le débit de données en exploitant, selon les releases, sur la modulation (64 QAM), sur l'utilisation d'antennes MIMO ou encore par l'utilisation de techniques de Dual Carrier par exemple.

Algorithme d'entrelacement pour le HSPA

Dans [HSP04] l'algorithme d'entrelacement du HSPA+ est présenté et les éléments suivant sont introduits :

K : Nombre de bits d'entrée de l'entrelaceur interne du Turbo-code.
R : Nombre de lignes de la matrice rectangulaire décrite dans le standard.
C : Nombre de colonnes de la matrice rectangulaire décrite dans le standard.
P : Nombre premier décrit dans le standard.
v: Racine primitive décrit dans le standard.

A partir de ces informations, le document présente l'algorithme à réaliser pour obtenir l'entrelacement des données.

- Calculer R de la matrice rectangulaire selon la formule suivante:

$$K \begin{cases} 5, \text{ si } (40 \leq K \leq 159) \\ 10, \text{ si } ((160 \leq K \leq 200) \text{ ou } (481 \leq K \leq 530)) \\ 20, \text{ si } (K = \text{autres valeurs}) \end{cases}$$

- Calculer la valeur de p et C, selon l'algorithme suivant :

```
1.  Début
2.  si (481≤K≤530)
3.      P=53
4.      C=p;
5.   sinon
6.      Déterminer p à partir du Tableau 5. 1  Liste des p nombres premiers et
        leurs racines primitivestel que K<R*(p+1),
7.      Calculer C de la matrice tel que :
8.      si (K≤R*(p-1) alors
9.         C=p-1
10.     sinon
11.        si (R*(p-1)<N<R*p)) alors
12.           C=p
13.        sinon
14.           si (R*p<N) alors
15.              C=p+1
16.           fin si
17.        fin si
18.     fin si
19.  fin Si
```

Ecrire la séquence de bits d'entrée dans la matrice rectangulaire colonne par colonne et si $R*C>K$, des bits supplémentaires sont utilisés pour remplir la matrice.

Tableau 5. 1 Liste des p nombres premiers et leurs racines primitives

p	v	p	v	p	v	p	v	p	v
7	9	47	5	101	2	157	5	223	3
11	2	53	2	103	5	163	2	227	2
13	2	59	2	107	2	167	5	229	6
17	3	61	2	109	6	173	2	233	3
19	2	67	2	113	3	179	2	239	7
23	5	71	7	127	3	181	2	241	7
29	2	73	5	131	2	191	19	251	6
31	3	79	3	137	3	193	5	257	3
37	2	83	2	139	2	197	2		
41	6	89	3	149	2	199	3		
43	3	97	5	151	6	211	2		

- Construire une séquence de base $S(j)$ pour une permutation intra-ligne :

$$S(j)=[\ v*S(j-1)\ ]\%\ p$$

avec $j=1,2,\ldots\ldots,p-2$

Calculer la dernière séquence des entiers premiers $q(i)$ pour $i=1,2\ldots,R-1$, en assignant $q(0)=1$, tel que $gcd(q(i), p-1) = 1$ et $q(i)>6$ et $q(i)>q(i-1)$

- Permuter la séquence $q(i)$ pour construire la séquence $r(i)$ tel que,

$$r_{T(i)}=q(i)\ avec\ i=0,1\ldots..,R-1$$

avec $T(i)$ une règle de permutation interligne définie dans le standard.

- Effectuer la permutation intra-ligne $U_i(j)$ tel que :

```
Début

1.  pour i =0,1,.....,R-1 et j =0,1,.....,p-2 faire
2.      si (C = p) alors
3.          Ui(j)=S[(j*r(i))mod(p-1)]
4.          Ui(p-1) = 0
5.      fin si
6.      si (C = p+1 ) alors
7.          Ui(j)=S[(j*r(i)) , Ui(p-1) = 0
8.          Ui(p) = p
9.      fin si
10.     si (K = R*C) alors
11.         Echanger UR-1(p) avec UR-1(0)
12.     fin si
13.     si (C = p-1) alors
14.         Ui(j)=S[(j*r(i)) mod (p-1)]-1
15.     fin si
16. fin pour

Fin
```

- Effectuer la permutation interligne de la matrice basée sur *T(i)* avec *T(i)* la position de la ligne d'origine de la $i^{éme}$ ligne permutée et définie dans le standard.
- Lire les bits colonne par colonne dans la matrice rectangulaire après avoir supprimé les bits supplémentaires rajoutés à la séquence des bits d'entrée.

Nous détaillons par la suite les différentes étapes de cet algorithme à travers un exemple pédagogique (*K =40) ;* d'après les définitions introduites précédemment : *R=5, C=10, p=11, v=2.*

L'étape suivante consiste à remplir une matrice d'ordre r*10 par les 44 données à partir de la ligne 0. Puisque la matrice contient 50 cellules, les cellules vides restantes sont remplies par des bits supplémentaires représentés par la valeur -1 dans la dernière ligne (voir Figure 5. 4).

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	-1	-1	-1	-1	-1	-1

Figure 5. 4 *Arrangement de K=44 données en une matrice 5*10*

Ensuite les séquences de valeurs *S, Q, R* sont calculées en se basant sur les règles définies dans le standard. :

S = 1 2 4 8 5 10 9 7 3, tel que le nombre de valeurs dans S soit *(p-1)= 10*
Q = 1 7 11 13 17, tel que le nombre de valeurs dans Q soit *R =5*
R = 17 13 11 7 1, tel que le nombre de valeurs dans R soit *Q=5*

Valeurs de U :

0 6 4 1 2 9 3 5 8 7
0 7 8 5 3 9 2 1 4 6
0 1 3 7 4 9 8 6 2 5
0 6 4 1 2 9 3 5 8 7
0 1 3 7 4 9 8 6 2 5

Tel que le nombre de valeurs dans U est *R*C = 5*10=50*.

Les valeurs dans U sont ensuite utilisées pour effectuer une permutation intra-ligne. Les valeurs U de la première ligne sont utilisées pour permuter la première ligne de la matrice. Ainsi concernant notre exemple, en suivant la première ligne de U on obtient : la première valeur de la trame garde sa place, la seconde valeur et la sixième valeur sont permutées, la troisième et la quatrième valeur sont permutées et ainsi de suite. La matrice obtenue après la permutation intra-ligne est illustrée dans (la Figure 5. 5).

0	3	4	6	2	7	1	9	8	5
10	17	16	14	18	13	19	11	12	15
20	21	28	22	24	29	27	23	26	25
30	33	34	36	32	37	31	39	38	35
40	41	-1	42	-1	-1	-1	43	-1	-1

Figure 5. 5 *Matrice après la permutation intra-ligne*

La dernière étape consiste à effectuer une permutation interligne sur la matrice rectangulaire en utilisant la permutation définie dans le standard. Concernant notre exemple, la permutation interligne est définie par :

T = 4, 3, 2, 1, 0

avec le nombre de valeurs dans T donné par *R=5*.

La matrice après la permutation interligne est illustrée dans la Figure 5. 6

40	41	-1	42	-1	-1	-1	43	-1	-1
30	33	34	36	32	37	31	39	38	35
20	21	28	22	24	29	27	23	26	25
10	17	16	14	18	13	19	11	12	15
0	3	4	6	2	7	1	-9	8	5

Figure 5. 6 *Matrice après la permutation interligne*

Après avoir enlevé les bits supplémentaires, les valeurs dans la matrice sont lues colonne par colonne afin de construire l'ordre entrelacé des données. L'ordre entrelacé obtenu pour *K*= 44 est alors le suivant,

π = 40 30 20 10 0 41 33 21 17 3 34 28 16 4 42 36 22 14 6 32 24 18 2 37 29 13 7 31 27 19 1 43 39 23 11 9 38 26 12 8 35 25 15 5

B. Résultat expérimentaux de l'approche d'assignation mémoire

Au travers de cette expérience nous voulons mettre en avant l'impact de la contrainte de réseau sur le coût de l'architecture finale. Pour ce faire, nous avons appliqué notre méthode sans optimisation du

contrôle dans un premier temps (cf. chapitre 3) à l'entrelaceur HSPA en considérant deux types de réseau d'interconnexion i.e. un réseau papillon et un barrel-shifter, tout en conservant des configurations identiques (tailles de trames…).

Le Tableau 5. 2 illustre la surface en nombre de slices des différents réseaux cibles implémentés en considérant les deux mêmes configurations du standard HSPA (i.e. 1024 données, parallélismes de traitement de 4 et 8 calculs parallèles). Comme nous l'avons mentionné dans les descriptions de notre architecture, les signaux/mots de contrôle sont stockés dans des ROMs dédiées.

Tableau 5. 2 *Surface en nombre de slices des différents réseaux d'interconnexions ciblés*

Standard	Nombre de données	Parallélisme	Barrel Shifter		Papillon		Barrière de MUX	
			Réseau	*Contrôle*	*Réseau*	*Contrôle*	*Réseau*	*Contrôle*
HSPA	*1024*	*4*	24	5	30	12	24	72
HSPA	*1024*	*8*	48	8	90	36	96	245

Comme mentionné par les auteurs de [SAN12], le coût du réseau est relativement négligeable comparativement au coût global des architectures. De plus, nous avons ici la confirmation qu'un réseau barrel shifter est moins couteux qu'un réseau papillon pour le réseau et pour le contrôle. Nous verrons par la suite que cette « classification » peut être utilisée efficacement pour proposer au concepteur des solutions architecturales optimisées via un flot d'exploration généralisé.

Toutefois, si le réseau d'interconnexion peut sembler avoir un impact limité si on le compare l'architecture globale du système (cf. Tableau 5.3), l'expérience montre qu'il a un impact majeur sur le coup final du système.

Tableau 5. 3 *Etude de différents réseaux d'interconnexion*

Standard	Nombre de données	Parallélisme	Contrainte Réseau	Nb Registres	Slices registre	Slices LUT	Total Slices
HSPA	*1024*	*4*	*barrel shifter*	**427**	5606	7852	**10460**
HSPA	*1024*	*8*	*barrel shifter*	**709**	10274	31390	**36644**
HSPA	*1024*	*4*	*papillon*	**68**	885	2267	**2671**
HSPA	*1024*	*8*	*papillon*	**108**	1363	4454	**5141**

Le Tableau 5. 3 montre les résultats obtenus avec pour contrainte de réseau un papillon, puis un barrel shifter pour le cas d'un entrelaceur HSPA. Ces expérimentations ont été réalisées avec l'approche de placement mémoire colonne par colonne. L'objectif ici est de constater que si en effet un réseau d'interconnexion de type barrel-shifter se révèle dans l'absolu moins couteux (cf. Tableau 5. 3), dans la pratique, une telle contrainte de réseau peut se révéler très éloignée de la règle d'entrelacement (ce qui est le cas dans notre expérience). De ce fait, on constate qu'une grande quantité de registres additionnels est ajoutée (jusqu'à 709 registres pour un parallélisme 8). Ainsi, avec une contrainte réseau moins forte (réseau papillon) seuls 108 registres sont ajoutés.

Nous avons donc ici un exemple démontrant que l'adéquation de la règle d'entrelacement avec la contrainte de réseau définie par l'utilisateur est un élément important à considérer pour l'analyse des résultats, mais surtout à terme pour guider le concepteur dans le choix du réseau d'interconnexion cible qu'il choisira. S'assurer de cette adéquation nécessite donc une méthode dédiée à l'exploration automatique des solutions possibles.

C. Intérêt notre approche (placement mémoire et adressage)

Nous avons ensuite appliqué notre approche de placement mémoire et d'optimisation de l'architecture de contrôle (cf. Chapitre 4) à l'entrelaceur HSPA pour différents configurations (tailles de trames, parallélismes de traitement). Dans les exemples que nous proposons, nous ciblons un réseau de type Benes (toutes les permutations sont possibles) et nous appliquons l'approche de placement mémoire avec optimisation du contrôle.

Tableau 5. 4 *Liste des configurations testées (standard HSPA)*

	Standard	**Nb bits**	**Parallélisme**
Config1	HSPA	1024	4
Config2	HSPA Butterfly	1024	8
Config3	HSPA Butterfly radix4	1024	16
Config4	HSPA Radix 4	1024	8
Config5	HSPA	5120	8
Config6	HSPA Butterfly	5120	16
Config7	HSPA Butterfly radix4	5120	32
Config8	HSPA Radix 4	5120	16

En effet, il ne suffit pas d'être capable de supporter une contrainte réseau définie par l'utilisateur. Les expérimentations que nous avons menées montrent qu'en comparaison à une approche n'optimisant pas le coût du contrôleur (i.e. [CHA10a]), dans cette série d'expérimentations puisque le réseau obtenu sera identique pour [CHA10a] et notre méthode, les différences qui seront mise en exergues seront directement la conséquence de nos approches d'optimisation du contrôle.

Le tableau 5.4 présente les différentes configurations que nous avons retenues pour ces expérimentations.

Tableau 5. 5 *Application de notre approche*

Cas	[CHA10a]			Notre approche				Gain (en %)		
	Slice registre	Slice LUTs	**Total Slices**	Nb registres	Slice registre	Slice LUTs	**Total Slices**	Slice registre	Slice LUTs	**Total Slices**
Config1	137	834	**880**	0	135	490	**535**	*1,46*	*41,25*	***39,21***
Config2	233	1096	**1157**	0	230	809	**883**	*1,29*	*26,19*	***23,69***
Config3	429	1783	**1917**	0	532	2122	**2211**	*-24,01*	*-19,02*	***-15,34***
Config4	233	1094	**1154**	0	232	824	**898**	*0,43*	*24,69*	***22,19***
Config5	293	6590	**6705**	0	282	3643	**3747**	*3,76*	*44,72*	***44,12***
Config6	537	9273	**9448**	0	546	7023	**7167**	*-1,68*	*24,27*	***24,15***
Config7	1029	14545	**14864**	0	1071	13282	**13552**	*-4,09*	*8,69*	***8,83***
Config8	537	9194	**9361**	0	690	9028	**9142**	*-28,5*	*1,81*	***2,34***

D'après les résultats du Tableau 5. 5, grâce à notre approche combinant relaxation et adressage mémoire, le gain en termes de surface comparée aux résultats obtenus en appliquant [CHA10a] varie entre 2.39% et 44.11% (excepté dans le cas de la Config 3). Nous constatons également que dans la majorité des cas, ce gain s'améliore quand le rapport entre la taille du mot de code et le parallélisme est élevé. En effet, la régularité d'adressage peut être potentiellement favorisée dans une telle situation. En outre, plus le degré de parallélisme est grand, plus le coût de la logique d'aiguillage entre les RAMs et ROMs est potentiellement élevé.

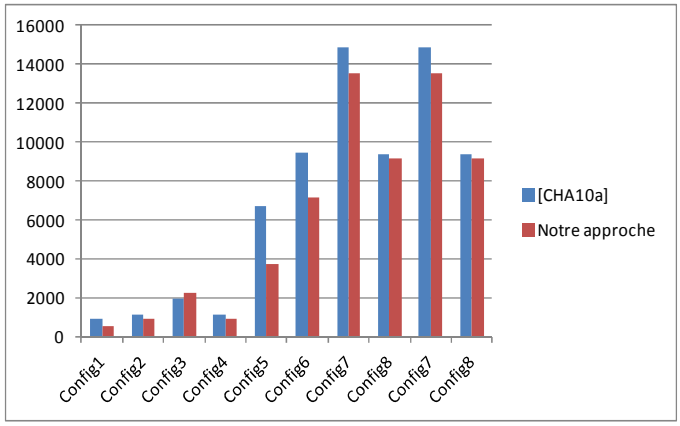


Figure 5. 7 *Comparaison en nombre de slices des architectures générées pour l'entrelaceur HSPA*

Dans le cas de la Config3, [CHA10a] génère un résultat plus optimisé par rapport à notre approche. En analysant les rapports de synthèse, il s'avère que le coût des signaux contrôlant l'accès aux RAMs et la logique d'aiguillage ajoutée est supérieur au gain obtenu suite à l'optimisation des ROMs.

De façon plus générale, ce que nous observons ici confirme notre hypothèse de départ : pour pouvoir générer des architectures optimisées il faut être capable de respecter une contrainte de réseau, mais également d'optimiser la partie contrôle de l'architecture, problématique que les approches de l'état de l'art, et notamment [CHA10a] ne prennent pas en compte.

III. Entrelaceur pour l'Ultra-Wide Band

Une seconde campagne expérimentale a été menée afin de confronter notre approche à une seconde famille d'entrelaceur en travaillant sur le standard Ultra Wide Band [IEE07].

A. Présentation

Ces dernières années, les systèmes de communication large bande (Ultra-Wide Band, UWB) ont fait l'objet d'une grande attention de la part de l'industrie et de la recherche académique. Les raisons sont que ces technologies peuvent permettre d'atteindre des débits très importants, à un coût réduit en termes de consommation et de surface. Plusieurs systèmes OFDM (pour **O**rthogonal **F**requency **D**ivision **M**ultiplexing, [FLO95]) ont ainsi été proposés pour implémenter des systèmes UWB par exemple [IEE07].

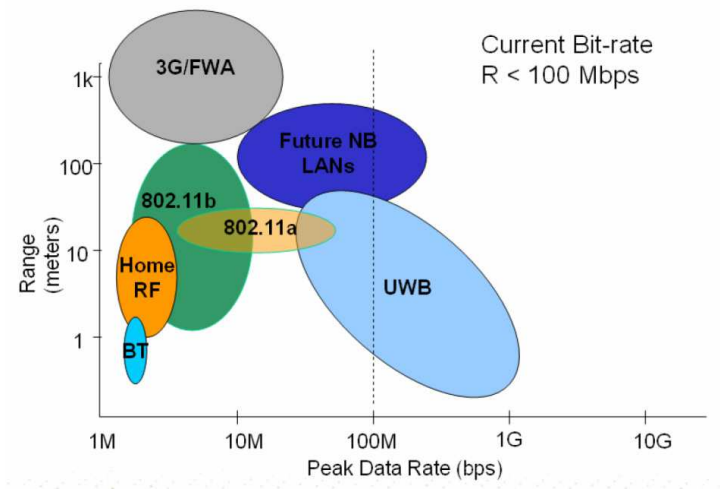


Figure 5. 8 *Les différents systèmes WLAN /PLAN existant*

L'OFDM est une modulation de signaux numériques par répartition en fréquences orthogonales. Le principe de l'OFDM consiste à diviser sur un grand nombre de porteuses le signal numérique que l'on veut transmettre. Pour que les fréquences des porteuses soient les plus proches possibles et ainsi transmettre le maximum d'information sur une portion de fréquences donnée, l'OFDM utilise des porteuses orthogonales entre elles. Les signaux des différentes porteuses se chevauchent mais grâce à l'orthogonalité n'interfèrent pas entre elles.

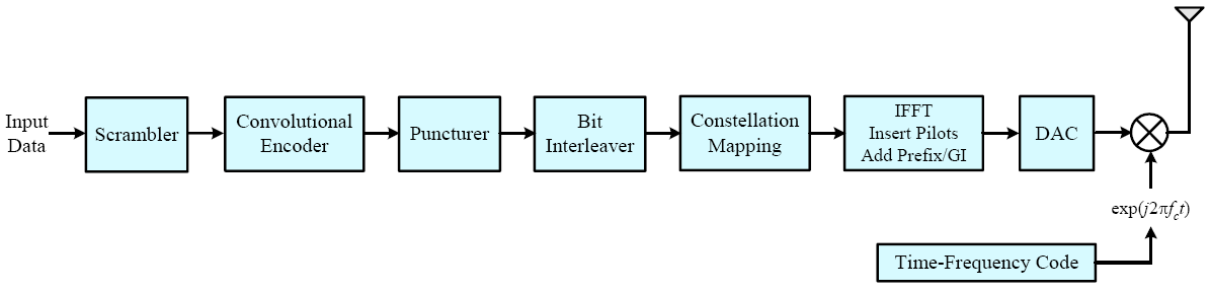


Figure 5. 9 *Schéma bloc d'une architecture OFDM pour un émetteur*

C'est ce type de système qui est exploité dans le cadre de la norme WPAN-802.14.3a (Wireless Personal Area Network, [IEE07]). Comme nous pouvons le constater (voir Figure 5. 9), cette architecture requière l'utilisation d'un entrelaceur. La règle d'entrelacement utilisée est définie par les relations mathématiques suivantes :

$$S(i) = U\left\{\text{Floor}\left(\frac{i}{N_{CBPS}}\right) + (6/TSF) * \text{Mod}(i, N_{CBPS})\right\}$$

$$T(i) = S\left\{\text{Floor}\left(\frac{i}{N_{T\,\text{int}}}\right) + 10\text{Mod}(i, N_{Tint})\right\}$$

Les symboles NCBPS, NTint, et TSF font respectivement référence au nombre de bits codés par symbole, à la taille d'un symbole et au facteur de dispersion temporel (Time Spreading Factor).

Le principe de l'entrelacement est le suivant :

- Un entrelacement des symboles entre eux,
- Un entrelacement des bits au sein d'un symbole.

Au final, en sortie de l'entrelaceur, les bits ont été entrelacés dans les symboles et entre les différents symboles. L'entrelaceur que nous ciblons doit pouvoir être utilisé dans trois modes de fonctionnement différents en fonction de la longueur de la trame à entrelacer (300, 600 ou 1200 données).

B. Résultats expérimentaux

Nos premières expérimentations nous ont amenées à la conclusion que les contraintes de réseau d'interconnexion avaient un impact important sur le coût des architectures générées, et que donc disposer d'une approche capable de prendre en compte cette contrainte était nécessaire. Toutefois, ainsi que le laisse apparaitre le tableau 5.5, il faut être capable de le faire de la façon la plus efficace possible, c'est ce que nous voulons observer avec ce nouveau cas d'étude.

Nous avons donc appliqué notre approche d'assignation et d'adressage mémoire (cf. chapitre 4) à l'entrelaceur UWB pour les trois modes de fonctionnement précités (i.e. 300, 600 et 1200 données) et pour différents degrés de parallélisme. Dans une première expérience, nous analysons l'apport de l'approche méthodologique présentée dans le chapitre 4 (optimisation du contrôle par fusion des ROM) en comparaison de l'approche de placement mémoire présentée dans le chapitre 3 (placement mémoire avec exploration de l'espace de conception données par données). Puis dans une seconde série d'expériences nous comparons les résultats que nous obtenons avec ceux de [CHA10a] pour différents réseaux d'interconnexion cibles (i.e. une barrière de multiplexeurs et un barrel shifter).

Tableau 5. 6 *Liste des configurations de l'entrelaceur UWB*

	Nb données	**Parallélisme**
Config1	300	3
Config2	300	4
Config3	300	8
Config4	600	3
Config5	600	4
Config6	600	8
Config7	1200	3
Config8	1200	4
Config9	1200	8

Nous avons dans un premier temps exploré l'espace des solutions de placement mémoire en comparant l'approche de placement mémoire sans optimisation du contrôle (cf. chapitre 3, exploration donnée par donnée) avec l'approche de placement mémoire avec optimisation du contrôle du chapitre 4. Ces premières expérimentations ont étés menées en ciblant un réseau d'interconnexion « barrel shifter ».

Tableau 5. 7 *Analyse de l'apport de l'optimisation du contrôle*

Cas	Placment mémoire sans optimisation du contrôle				Placment mémoire avec optimisation du contrôle				Gain (en %)		
	Nb registres ajoutés	Slice registre	Slice LUT	**Total Slices**	Nb registres ajoutés	Slice registre	Slice LUT	**Total Slices**	Slice registre	Slice LUT	**Total Slices**
Config1	0	103	277	**311**	0	69	155	185	*33,01*	*44,05*	***40,52***
Config2	71	1244	1747	**2309**	71	1239	1706	2254	*0,41*	*2,35*	***2,39***
Config3	154	2452	4296	**5509**	154	2435	4281	5472	*0,7*	*0,35*	***0,68***
Config4	47	775	1600	**1951**	47	757	1354	1692	*2,33*	*15,38*	***13,28***
Config5	222	2652	4229	**5523**	222	2682	3862	5156	*-1,14*	*8,68*	***6,65***
Config6	307	9507	17557	**22577**	307	5258	10172	12773	*44,7*	*42,07*	***43,43***
Config7	209	3459	5940	**7685**	209	3238	5763	7187	*6,39*	*2,98*	***6,49***
Config8	314	4225	6929	**8851**	314	2682	3682	5156	*36,53*	*46,87*	***41,75***
Config9	647	9681	19934	**24779**	647	9666	18933	23933	*0,16*	*5,03*	***3,42***

Les résultats obtenus montrent l'intérêt de l'approche de placement mémoire avec optimisation du contrôle. Les résultats montrent des gains pouvant aller jusqu'à 43% de la surface. Toutefois, on constate que l'essentiel des gains affichés pour ces cas d'étude et inférieur à 10%. Ces faibles gains sont probablement induits par la contraint de réseau.

Une nouvelle série d'expériences a été menée en modifiant la contrainte de réseau (en ciblant un réseau d'interconnexion non contraint (i.e. barrière de multiplexeurs). Nous comparons également les résultats avec ceux obtenus en utilisant une approche de placement mémoire de l'état de l'art, [CHA10a].

Tableau 5. 8 *Comparaison de notre approche avec [CHA10a] pour l'entrelaceur UWB*

Cas	[CHA10a]			Placment mémoire avec optimisation du contrôle				Gain (en %)		
	Slice registre	Slice LUT	**Total Slices**	Nb registres ajoutés	Slice registre	Slice LUT	**Total Slices**	Slice registre	Slice LUT	**Total Slices**
Config1	102	310	**346**	**0**	69	155	**185**	*32,36*	*50*	***46,54***
Config2	125	449	**501**	**0**	120	267	**311**	*4*	*40,54*	***37,93***
Config3	261	614	**684**	**0**	222	532	**628**	*14,95*	*13,36*	***8,19***
Config4	112	644	**692**	**0**	126	455	**499**	*-12,5*	*29,35*	***27,9***
Config5	137	866	**924**	**0**	134	546	**599**	*2,19*	*36,96*	***35,18***
Config6	233	1052	**1139**	**0**	229	688	**773**	*1,72*	*34,61*	***32,14***
Config7	199	1454	**1532**	**0**	211	954	**1027**	*-6,04*	*34,39*	***32,97***
Config8	149	1374	**1438**	**0**	145	899	**956**	*2,69*	*34,58*	***33,52***
Config9	248	1971	**2068**	**0**	248	1291	**1386**	*0*	*34,51*	***32,98***

En comparaison des expérimentations synthétisées dans le tableau 5.7, on constate des réductions de surface de plusieurs ordres de grandeur. De plus, les résultats de synthèses obtenus (cf. Tableau 5. 8) montrent des optimisations, par rapport à [CHA10a], variant entre 8,1% et 46.5% de la surface. En moyenne, notre approche permet d'optimiser la surface totale de 28,25%. Ceci démontre que la solution que nous proposons permet d'obtenir des gains très importants.

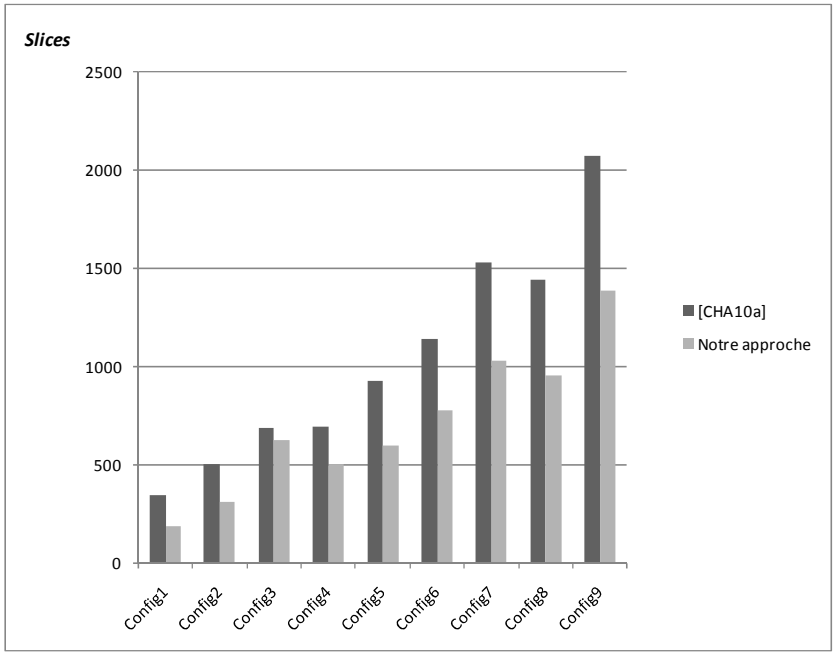


Figure 5. 10 *Comparaisons des résultats obtenus pour un entrelaceur UWB*

Concernant la contrainte de réseau cible « barrel shifter » (cf Tableau 5. 9), il est important de noter que l'approche [CHA10a] n'est pas capable de cibler un réseau d'interconnexion particulier. En outre, nous constatons qu'en imposant le barrel shifter comme contrainte de réseau, le surcoût en registres peut-être relativement élevé (cf. Figure 5. 11). De fait, le coût des registres additionnels croit avec le parallélisme. Ainsi, si l'on compare les résultats que nous obtenons avec pour cible un barrel shifter (avec optimisation du contrôle), avec les architectures obtenues à l'aide de [CHA10a] (avec un réseau de type barrière de multiplexeurs) seule la configuration 1 s'avère pertinente, le gain est alors de 46.53%. De fait, cette configuration est parfaitement adaptée à l'utilisation d'un barrel shifter comme réseau d'interconnexion (aucun registre additionnel).

Tableau 5. 9 *Comparaison de notre approche avec [CHA10a] pour l'entrelaceur UWB*

Cas	[CHA10a]			Placment mémoire avec optimisation du contrôle				Gain (en %)		
	Slice registre	Slice LUT	**Total Slices**	Nb registres ajoutés	Slice registre	Slice LUT	**Total Slices**	Slice registre	Slice LUT	**Total Slices**
Config1	102	310	**346**	**0**	69	155	**185**	*32,36*	*50*	***46,54***
Config2	125	449	**501**	**71**	1239	1706	**2254**	*-891,2*	*-279,96*	***-349,91***
Config3	261	614	**684**	**154**	2435	4281	**5472**	*-832,96*	*-597,24*	***-700***
Config4	112	644	**692**	**47**	757	1354	**1692**	*-575,9*	*-110,25*	***-144,51***
Config5	137	866	**924**	**222**	2682	3862	**5156**	*-1857,67*	*-345,96*	***-458,01***
Config6	233	1052	**1139**	**307**	5258	10172	**12773**	*-2156,66*	*-866,93*	***-1021,43***
Config7	199	1454	**1532**	**209**	3238	5736	**7187**	*-1527,14*	*-294,5*	***-369,13***
Config8	149	1374	**1438**	**314**	2682	3682	**5156**	*-1700*	*-167,98*	***-258,56***
Config9	248	1971	**2068**	**647**	9666	18933	**23933**	*-3797,59*	*-860,58*	***-1057,31***

Nous avons donc ici de nouveau la démonstration que si le concepteur choisi un réseau d'interconnexion cible trop contraint par rapport à la règle d'entrelacement qu'il souhaite implémenter, le surcoût peut-être très élevé.

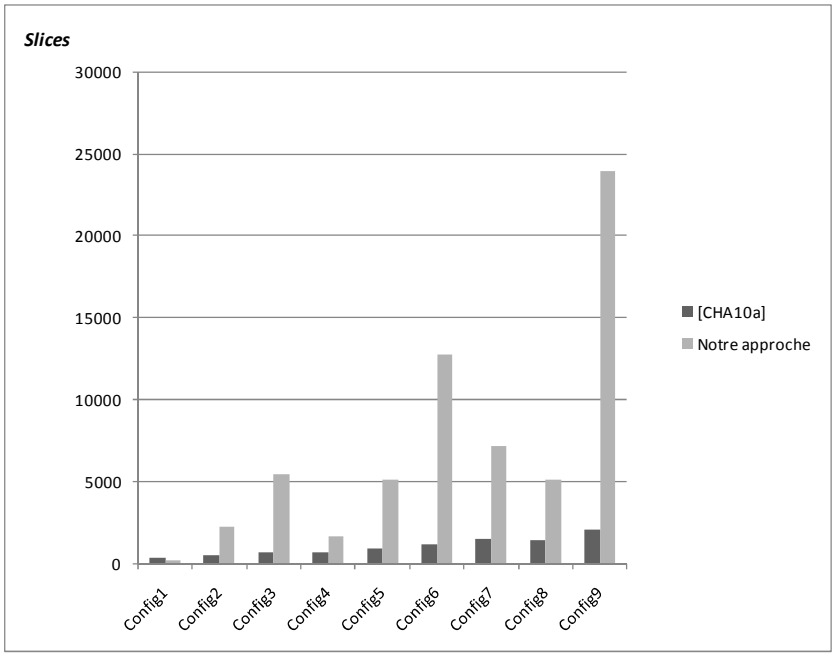


Figure 5. 11 *Comparaisons des résultats pour un entrelaceur UWB en ciblant un barrel shifter*

Enfin, on peut noter que cette série d'expérience démontre que déterminer le bon réseau d'interconnexion cible peut avoir un impact conséquent sur les gains que nous pouvons espérer atteindre en utilisant seule notre approche. Toutefois, déterminer le bon réseau d'interconnexion pouvant s'avérer une tâche ardue pour le concepteur, un flot de conception dédié s'avère nécessaire si l'on souhaite explorer le plus efficacement possible l'espace des solutions.

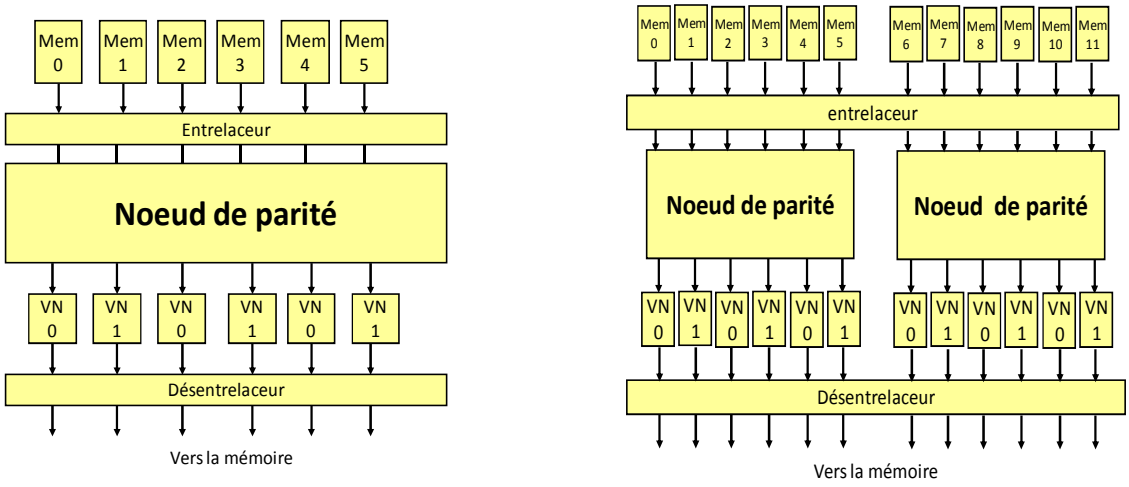
IV. Conception d'architectures de décodeurs LDPC

A. Les codes LDPC aléatoires

Les codes LDPC aléatoires sont définis avec des matrices creuses (une matrice creuse est une matrice avec un nombre de '1' très faible). Il a été montré que cette famille de code possède de performances asymptotiques très proches de la limite de Shannon [Sha48]. En outre, les performances des codes aléatoires sont relativement meilleurs que celles des codes structurés pour des codes à faible rendement. Les codes LDPC ont été d'abord présentés dans [GAL63] et redécouvert ensuite par MacKay [MAC99]. Les architectures typiques des décodeurs LDPC séries et partiellement parallèles sont illustrées dans la Figure 5. 12.

Dans les architectures séries de décodeur LDPC les nœuds de parités sont traités en séries, afin de réaliser un accès parallèle à la mémoire celle-ci est divisée en *B* bancs mémoire pour que le processeur puissent recevoir simultanément B messages. L'entrelaceur ainsi que le désentrelacer sont utilisés pour transférer les données entre le décodeur et les bancs mémoires. Le choix de *B* dépend du débit de l'application cible. Afin d'améliorer le débit, un ensemble nœuds de parité peut être traité simultanément dans les architectures partiellement parallèles, le nombre de processeurs est égale au nombre de nœuds de parités traités en parallèle. La mémoire de base est divisé en $B=y*dc$ nombre de

bancs mémoire afin de lire *y*dc* messages, avec *dc* = degré maximale de nœud de parité et *y* = nombre de nœud de parité traité en parallèles.



(a) Architecture série *(b) Architecture partiellement parallèle (y=2)*

Figure 5. 12 *Exemple d'une Architecture typique des codes LDPC avec dc=5*

B. Les codes LDPC structurés

Les codes LDPC structurés sont parmi les codes les plus utilisés dans la majorité des standards de télécommunication [WIF08] [WIM06] actuels. Afin de réaliser des débits élevés, ces codes sont implémentés en utilisant des architectures partiellement parallèles. Cependant les architectures partiellement parallèles souffrent des problèmes de conflits mémoire. La problématique consiste à assigner les matrices *Z* dans les bancs mémoire de telle sorte qu'aucun risque de conflit ne peut survenir. Nous détaillons par la suite les architectures partiellement parallèles pour les codes structurés à travers un exemple tiré du standard WIMAX.

Soient,

W= nombre de lignes =12,
Y = nombre de colonnes =24,
$d_{c,max}$*= degré maximal de nœuds de parité =7*,
taille de mot de code =WxZ= 768,
r = rendement de code =Y-W/Y =1/2

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
1	-1	94	73	-1	-1	-1	-1	-1	55	83	-1	-1	7	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
2	-1	27	-1	-1	-1	22	79	9	-1	-1	-1	12	-1	0	0	-1	-1	-1	-1	-1	-1	-1	-1	-1
3	-1	-1	-1	24	22	81	-1	33	-1	-1	-1	0	-1	-1	0	0	-1	-1	-1	-1	-1	-1	-1	-1
4	61	-1	47	-1	-1	-1	-1	-1	65	25	-1	-1	-1	-1	-1	0	0	-1	-1	-1	-1	-1	-1	-1
5	-1	-1	39	-1	-1	-1	84	-1	-1	41	72	-1	-1	-1	-1	-1	0	0	-1	-1	-1	-1	-1	-1
6	-1	-1	-1	-1	46	40	-1	82	-1	-1	-1	79	0	-1	-1	-1	-1	0	0	-1	-1	-1	-1	-1
7	-1	-1	95	53	-1	-1	-1	-1	-1	14	18	-1	-1	-1	-1	-1	-1	-1	0	0	-1	-1	-1	-1
8	-1	11	73	-1	-1	-1	2	-1	-1	47	-1	-1	-1	-1	-1	-1	-1	-1	-1	0	0	-1	-1	-1
9	12	-1	-1	-1	83	24	-1	43	-1	-1	-1	51	-1	-1	-1	-1	-1	-1	-1	-1	0	0	-1	-1
10	-1	-1	-1	-1	-1	94	-1	59	-1	-1	70	72	-1	-1	-1	-1	-1	-1	-1	-1	-1	0	0	-1
11	-1	-1	7	65	-1	-1	-1	-1	39	49	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0	0
12	43	-1	-1	-1	-1	66	-1	41	-1	-1	-1	26	7	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0

Figure 5. 13 *Matrice* H_{base} *du standard WIMAX avec taille de mot de code =576 et r=1/2*

La Figure 5. 14 illustre l'architecture partiellement parallèle proposée pour cet exemple, soit $B=d_{cmax}=7$. Ainsi chaque nœud de parité peut accéder à 7 données dans un seul cycle d'horloge. En outre, P=Z=32 de manière à ce que une matrice Z peut être traitée en un seul cycle

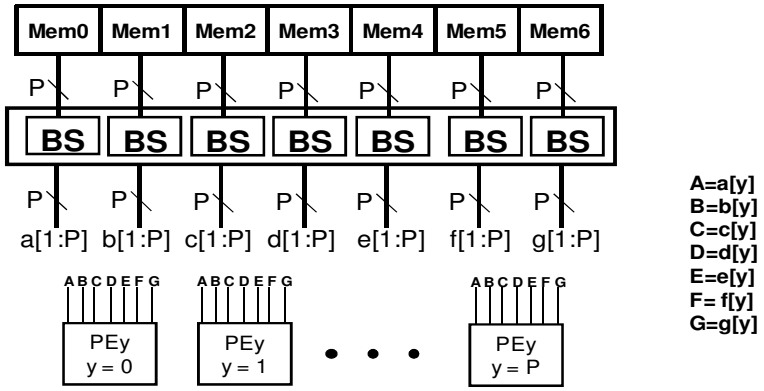


Figure 5. 14 *Architecture partiellement parallèle avec un code de rendement ½ pour le standard WIMAX*

La première étape consiste à préparer la matrice d'accès aux données. Cette matrice contient toutes les matrices *Z* qui doivent être accédées en parallèles. La matrice d'accès aux données pour cet exemple est illustrée dans Figure 5. 14. D'après la matrice *Hbase*, on peut voir par exemple qu'au premier cycle les *Z* matrices *2, 3, 9, 10, 13* et *14* sont accédées en parallèles, cela signifie que ces matrices doivent être stockées dans des bancs mémoires différents.

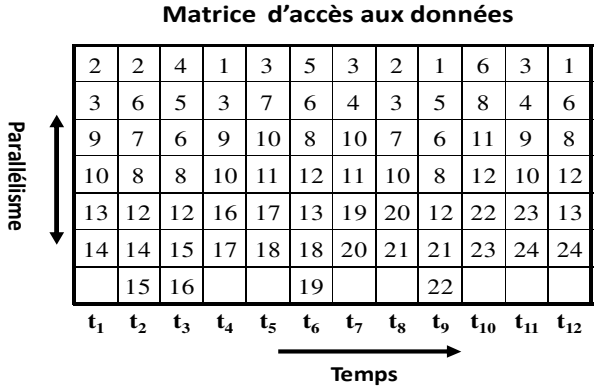


Figure 5. 15 *Matrice d'accès aux données utilisée pour le standard WIMAX*

Afin d'appliquer l'approche « d'allocation mémoire double », cette matrice d'accès aux données est convertie en une matrice d'allocation mémoire en divisant chaque colonne représentant un cycle d'horloge en deux semi-colonnes selon l'approche théorique que nous avons proposée dans le chapitre 3.

C. Résultats expérimentaux

Nous appliquons notre approche (cf. chapitre 4, placement mémoire avec optimisation du contrôle) au cas d'une architecture série d'un code LDPC aléatoire de taille *k=4376* bits avec *dc=degré maximal de nœud de parité=63* et *dv=degré maximal de nœuds de variables=4*. Ce code est construit en utilisant la méthode [MAC99]. Les tailles des bancs mémoire utilisés pour cette architecture série sont *B=16, 32*. Il est ici important de rappeler que notre approche, telle que nous l'appliquons ici, intègre la compaction des ROMs (minimiser les tailles de chaque ROM) mais pas encore la fusion des ROMs

(minimiser le nombre de ROMs), car l'implémentation de cette phase d'optimisation complémentaire n'est pas encore finalisée.

L'approche [CHA10a] ne peut pas être appliquée dans ce cas d'application puisqu'elle est basée sur « l'allocation mémoire simple » (i.e. elle fonctionne selon une technique dite de placement mémoire « in-place » qui ne peut s'appliquer que rarement dans le cas des codes LDPC, cf. chapitre 2 pour plus de détails). Nous comparons ainsi notre approche avec [CHA10]. Toutefois, nous rappelons que cette approche n'est pas capable de gérer une contrainte de réseau, nous ciblerons ici un réseau d'interconnexion non contraint (barrière de MUX). Nous nous plaçons ainsi dans un pire cas pour notre approche qui serait elle capable de cibler des réseaux d'interconnexion plus optimisés. De plus, nous constaterons que notre approche s'applique indifféremment pour les deux familles de LDPC que nous venons de décrire.

a) LDPC aléatoires

Les résultats obtenus montrent que notre approche est capable d'obtenir un gain en surface moyen de 13.22% par rapport à [CHA10] pour un parallélisme de traitement de 16. A l'inverse, on observe un surcoût de 7.18% dans le cas d'un parallélisme de 32. Comme expliqué précédemment, ce surcoût est dû à la logique d'aiguillage supplémentaire permettant d'aiguiller les signaux de contrôle issus des ROMs vers les différentes RAMs.

Tableau 5. 10 *Comparaison d'architectures séries d'un décodeur LDPC aléatoire*

Application	Nombre de données	Parallélisme	Contrainte Réseau	[CHA10]			Notre approche				Gain (en %)		
				Slice registre	Slice LUTs	Total Slices	Nb registres ajoutés	Slice registre	Slice LUTs	Total Slices	Slice registre	Slice LUTs	**Total Slices**
Gallager	*4376*	*16*	-	573	15171	**15373**	0	598	13143	**13342**	*-4,37*	*13,37*	***13,22***
Gallager	*4376*	*32*	-	1097	25738	**26086**	0	1157	28799	**27957**	*-5,47*	*-11,9*	***-7,18***

Afin d'analyser ce surcoût nous avons donc synthétisé séparément la partie du contrôleur dédiée à cette logique d'aiguillage induite afin de déterminer son impact sur le coût architectural total. Cette étude, Figure 5. 16 montre que cette logique représente selon les cas entre 15% et 30% de la surface totale de l'architecture.

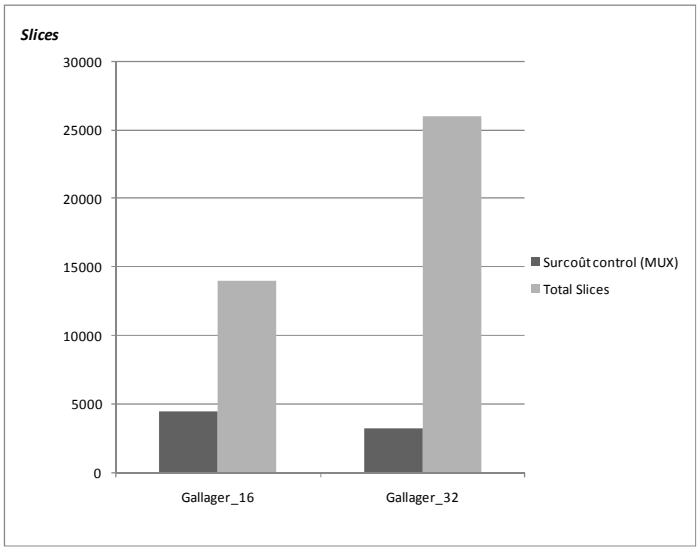


Figure 5. 16 *Surface de la partie contrôle des multiplexeurs supplémentaires rajoutés dans le cas de l'application de notre approche*

Nous sommes donc là face à un facteur pouvant dégrader la qualité de nos résultats. Pour répondre à cette problématique, l'approche de fusion des ROMs (chapitre 4) peut amener une solution pertinente. De fait, si moins de ROMs sont nécessaires, il en résultera directement les MUX utilisées seront de tailles plus petites et donc on aura moins de signaux de contrôle.

b) *LDPC structurés*

Nous avons exploré deux cas d'étude tirés du standard WIMAX :

- WIMAX ½ avec Z = P = 32, B =7,
- WIMAX 2/3 avec Z = P = 32, B =10.

Le tableau 5.10 illustre les résultats de surfaces en nombre de slices en appliquant notre approche ainsi que [CHA10] sur ces deux cas d'études. Le réseau d'interconnexion que nous ciblons est une barrière de multiplexeurs.

Tableau 5. 11 *Résultat en nombre de slices pour un entrelaceur WIMAX*

Application	Nombre de données	Parallélisme	Contrainte Réseau	[CHA10]			Notre approche				Gain (en %)		
				Slice registre	Slice LUTs	Total Slices	Nb registres ajoutés	Slice registre	Slice LUTs	Total Slices	Slice registre	Slice LUTs	Total Slices
WiMAX 1/2	*84*	*7*	-	128	191	**216**	0	127	221	**251**	*0,79*	*-15,71*	***-16,21***
WiMAX 2/3	*84*	*10*	-	161	454	**489**	0	154	454	**424**	*4,35*	*0*	***13,3***

Dans le cas de ce standard WIMAX notre approche représente un gain de 13.2% en nombre de slice dans la pour un rendement 2/3 et à l'inverse un surcoût matériel de 16,2% dans le cas d'une architecture avec un rendement ½. Ainsi avec un rendement 2/3, la loi de permutation combinée au parallélisme de traitement nous a permis d'extraire une régularité maximale de façon à ce que le gain obtenu était supérieur au coût de la partie contrôle rajouté.

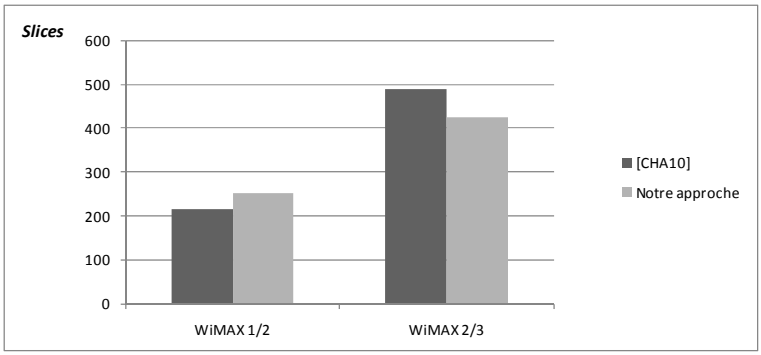


Figure 5. 17 *Architectures de l'entrelaceur WIMAX (réseau de type barrière de multiplexeurs)*

Nos comparaisons ont été faites par rapport à [CHA10] qui utilise le concept de « allocation mémoire double ». En effet, si nous nous étions comparé à [TAR04], puisque cette approche va nécessiter plus de mémoire (représentation des données en SSA), nos comparaisons n'auraient pas été honnêtes car les architectures obtenues avec [TAR04] supposent d'utiliser beaucoup plus de mémoire. Quoi qu'il en soit, notre approche reste plus riche que [TAR04] de part les avancées qu'elle amène par rapport à l'état de l'art : respect d'une contrainte de réseau pour toute règle d'entrelacement de données, optimisation du contrôle.

En conclusion de ces expériences on constate à nouveau que pour les LDPC, comme pour les Turbo Codes, le choix du réseau d'interconnexion cible est primordial pour obtenir une architecture finale à un coût raisonnable. Toutefois, ce choix s'avère à priori très complexe si l'on ne dispose pas d'outil

dédiée à cette exploration. C'est là que nos travaux en collaboration avec Saeed Ur Rehlan prennent tout leur sens.

V. Premières expérimentation un flot d'exploration généralisé

Ainsi que nous l'avons déjà évoqué, l'approche de placement mémoire basée sur la relaxation de la contrainte mémoire, combinée à l'optimisation du coût du contrôleur permet d'obtenir des gains significatifs en terme de coût architectural. Toutefois, comme nous l'avons constaté lors de ces expérimentations, si le concepteur souhaite obtenir une architecture basée sur un réseau d'interconnexion trop éloigné de ce que peut supporter nativement la règle d'entrelacement ciblée, le surcoût apporté par l'ajout de registres (et la logique d'aiguillage associée) peut limiter les gains observés.

Pour rappel, le principe est le suivant : le concepteur fournis un ensemble de contraintes (parallélisme, type de réseau cible, règle d'entrelacement) qui seront ensuite exploités par les algorithmes de placement mémoire. Un ensemble de réseau d'interconnexion possible est également fourni en entrée. Le principe est de laisser la liberté à l'utilisateur de déterminer s'il souhaite voir notre approche généralisée lui fournir un ensemble de solutions de placement mémoire possibles (en fonction des différents réseaux utilisables depuis la librairie), ou bien s'il est près à payer un éventuel surcoût pour avoir la certitude d'obtenir une architecture exploitant de façon certaine le réseau d'interconnexion qu'il a sélectionné. A partir de ces informations, notre approche va générer des placements de données sans conflits mémoires en exploitant les différentes approches de relaxation séparément (relaxation réseau, relaxation mémoire) ou en les combinant (relaxation réseau et mémoire). Un ensemble de solution est ainsi généré.

A titre d'exemple, pour un parallélisme donné le coût d'un réseau de type *papillon* est plus faible que le coût d'un *réseau de Benes.* De plus, si l'on prend pour base un réseau *papillon* et qu'on lui adjoint des registres (relaxation mémoire) ou des permutations élémentaires (relaxation réseau), ou les deux, le coût de l'architecture correspondante augmente. Donc, la limite supérieure qui permettra de stopper l'exploration des solutions de relaxation pour un réseau de type *papillon* sera la suivante : si le coût du *papillon* avec relaxation est supérieur ou égal au coût d'un *réseau de Benes*, alors il n'est plus possible d'optimiser l'architecture.

Pour ce faire, suite à une étude des complexités de résultats de synthèses, nous nous basons sur les comparaisons suivantes en termes de coût architectural pour un parallélisme donné :

Barrel Shifter **<** *Papillon* **<** *Réseau de Benes* **<** *Cross Bar*

Premier résultats expérimentaux

Le flot proposé n'est à ce jour pas complètement réalisé, en particulier l'optimisation du contrôleur et la compaction des ROMs proposées dans cette thèse ne sont pas intégrées, et la génération du code VHDL final n'est pas fonctionnelle. Les résultats que nous présentons ici sont donc issus d'estimations exploitant les mêmes approches méthodologique et technologique que dans [SAN12] (90nm de STMicroelectronics).

Ces premières expérimentations, dont nous montrons ici quelques échantillons représentatifs, sont menées sur un cas d'étude HSPA. Toutes les longueurs de trames de standard ont étés explorées, nous présentons ici un cas avec 40, un autre avec 800 et un dernier avec 2240 données. Nous présentons le coût des multiplexeurs (MUX), du réseau, des registres additionnels et le coût total de l'architecture et ce pour un réseau de type barrel shifter (BS), un papillon (BF), un réseau de Benes, un cross bar (CB), pour un réseau ad hoc, ou *custom*, généré par notre flot généralisé (NWC), un barrel shifter enrichi par

des permutations élémentaires issues de la relaxation du réseau (BS+NWC), un barrel shifter enrichi par des registres issus de la relaxation mémoire (BS+M) et enfin une architecture combinant relaxation mémoire et réseau (BS+NWC+M).

Bien que ces résultats (cf. Figure 5. 18) ne tirent pas encore profit des optimisations que nous proposons dans nos travaux, ils montrent toutefois la pertinence des solutions que nous proposons. Nous pouvons ainsi observer que globalement, pour tout ces cas d'application il est possible d'utiliser un réseau de type papillon.

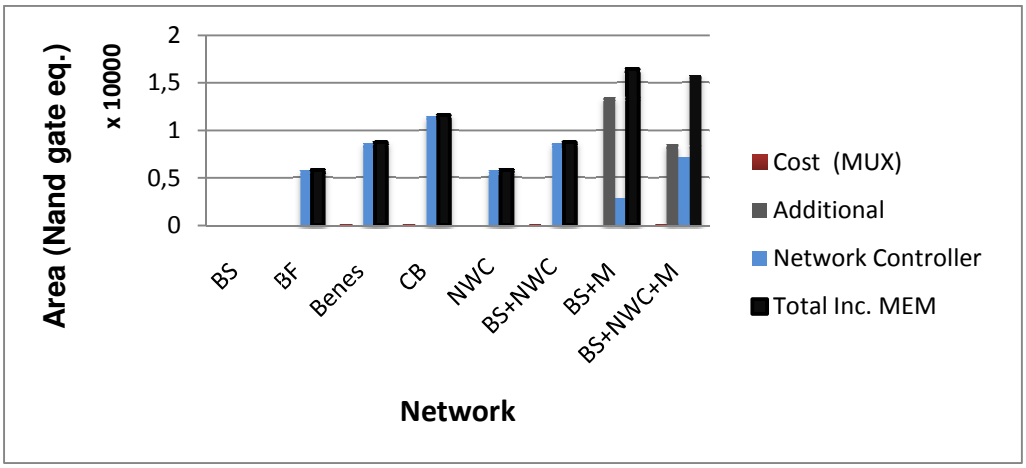


Figure 5. 18 *HSPA 40 data*

On constate que notre flot général est capable de générer automatiquement un réseau ad hoc (NWC) et coût équivalent à un réseau de type papillon. Puisque nous cherchons à offrir un ensemble de solutions possibles, allons chercher donc à déterminer si à partir d'un réseau moins complexe (ici un barrel shifter), il est possible d'obtenir une architecture moins coûteuse en appliquant nos principes de relaxation.

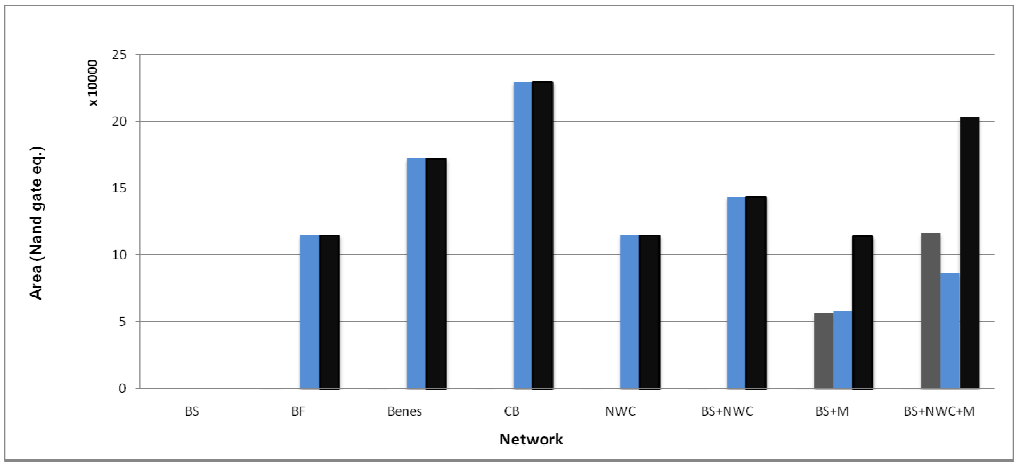


Figure 5. 19 *HSPA 800 data*

Puisqu'en l'état actuel ce flot généralisé n'intègre pas toutes les contributions théoriques que nous proposons dans cette thèse pour optimiser le coût du contrôleur, nous constatons que le coût de l'architecture finale reste élevé lorsque l'on cherche à combiner les deux axes de relaxation (mais

inférieur au coût d'un cross bar). Toutefois prise individuellement les techniques de relaxation montrent des résultats qui vont dans le bon sens (e.g. pour 800 données, le coût de l'architecture obtenue avec uniquement la relaxation mémoire est inférieur au coût d'un papillon).

Il est naturellement impossible de tirer des conclusions définitives en l'état puisque que nous n'utilisons qu'un prototype de l'outil et que celui-ci n'intègre qu'une partie de nos approches. Toutefois, les résultats que nous obtenons pour l'instant montrent que nous allons dans le bon sens.

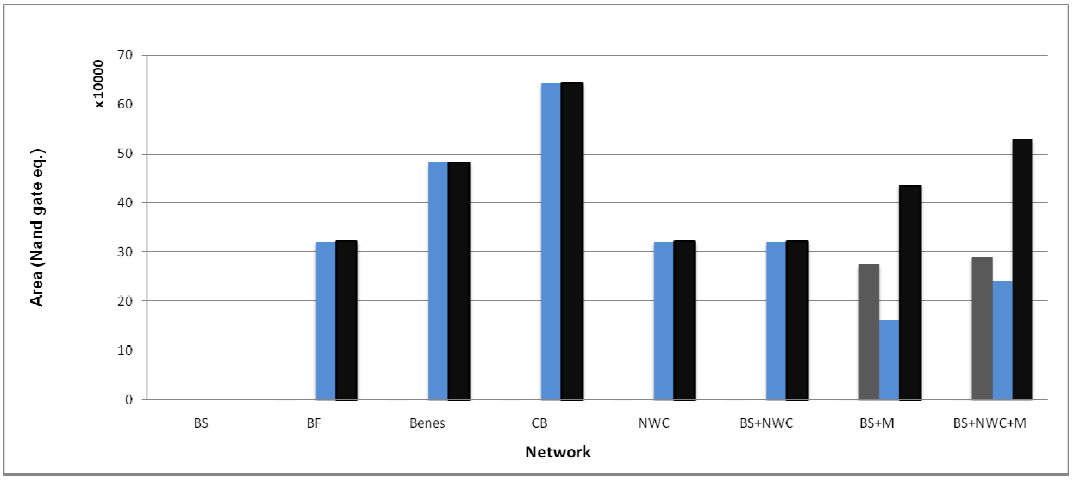


Figure 5. 20 *HSPA 2240 data*

Autre axe d'optimisation possible

La méthode d'allocation mémoire ainsi d'adressage mémoire que nous proposons est basée sur le concept « d'allocation mémoire double », par conséquent chaque ROM contient les adresses des accès en écriture et en lecture aux mémoires. Néanmoins, dans le cas d'une allocation mémoire simple qui est utilisée dans la plupart des entrelaceurs Turbo-Codes. Le processeur va ainsi lire et écrire l'information extrinsèque dans la même adresse du même banc mémoire. Dans ce cas, au lieu de mémoriser deux adresses mémoires identiques pour chaque donnée (adresse de lecture et adresse d'écriture), il est possible de n'en mémoriser qu'une seule. Nous sommes donc là en présence d'un autre axe d'optimisation important pour des Turbo-Codes.

VI. Bilan

L'ensemble des expériences présentées dans ce chapitre a permis de mettre en avant l'intérêt de notre approche qui est capable de trouver un placement mémoire sans conflit sous contrainte de réseau tout en optimisant le coût du contrôleur.

Dans le cas des codes LDPC l'efficacité de nos algorithmes, en l'état, dépend des configurations données (taille de la trame, degré parallélisme, lois de permutation). Nos travaux montrent toutefois que l'intégration de l'approche de fusion des ROMs va amener des optimisations pertinentes et importantes compte tenu des nos observations. De plus, nos résultats expérimentaux montrent que dans la majorité des cas d'application Turbo Code, notre approche permet d'atteindre des gains considérables.

Enfin, les recherches menées avec S. Ur Rehman ouvrent la voie à un flot permettant d'obtenir un bon compromis entre la complexité du réseau, le surcoût en registres et le coût du contrôle. Ainsi, l'intégration de notre approche dans un nouveau flot d'exploration généralisé (cf chapitre 4) fait sens. Comme expliqué dans le chapitre précédent, ce flot explorera d'une manière exhaustive les différentes possibilités d'assignation mémoire toute en permettant de relâcher les contraintes de mémorisation (mémoriser certaines données dans des registres) et/ou de réseau (ajouter des multiplexeurs) afin de d'offrir un ensemble de solutions architecturales pertinentes au concepteur.

Conclusion et perspectives

Les Turbo-Codes et les codes LDPC sont au cœur des normes de télécommunication actuelles grâce à leurs excellentes capacités de correction d'erreurs. La complexité croissante des algorithmes mis en œuvre et l'augmentation continue des volumes de données et des débits applicatifs requièrent souvent la conception de circuits intégrés dédiés (ASIC). Aujourd'hui, la complexité et le coût de ces systèmes sont très élevés; les concepteurs doivent pourtant parvenir à minimiser la consommation et la surface total du circuit, tout en garantissant les performances temporelles requises. Pour répondre à ces défis, les concepteurs jouent notamment sur le parallélisme de traitement dans la mise en œuvre de l'application. Toutefois, cela suppose que les éléments de calcul qui fonctionnent en parallèle s'échangent des données efficacement i.e. qu'il n'y ait pas ou peu de conflits d'accès aux mémoires. Ainsi, comme nous l'avons vu, de nombreux travaux de recherche tentent de résoudre ces problèmes. Malheureusement, aucune des approches de l'état de l'art ne cherche à trouver un placement de données en mémoire supprimant les conflits d'accès tout en respectant une cible architecturale (i.e. un réseau d'interconnexion) et en optimisant le coût de l'architecture du contrôleur du système.

Dans ce contexte, nous avons présenté un ensemble d'approches de placement de données en mémoire répondant à ces défis. Dans un premier temps nous avons présenté deux méthodes capables pour toute cible de réseau d'interconnexion et pour toute règle d'entrelacement (même si elle est incompatible avec le réseau ciblé), de trouver un placement mémoire et exploitant le principe de *relaxation* de la contrainte mémoire (i.e. l'ajout de registres additionnels).

Dans un second temps, nous avons présenté une évolution de la deuxième méthode. Cette nouvelle approche permet d'optimiser l'architecture du contrôleur de l'application, en prenant en considération, dès l'étape de placement mémoire, la problématique du coût du pilotage des mémoires dans lesquels sont assignées les données. Pour ce faire un modèle de graphe dédié à la formalisation des conflits d'adressage entre les données (ACG) a été défini. Le flot de conception exploitant ce modèle a ensuite été formellement présenté.

Nous avons enfin présenté une série d'expérimentations montrant la pertinence des solutions adoptées pour les algorithmes de placement de données en mémoire sans conflit via relaxation de la contrainte mémoire. Nous présentons plusieurs cas d'études visant à montrer le potentiel de nos approches.

Enfin, dans une dernière partie nous avons introduit une approche plus globale permettant d'explorer et d'optimiser le coût totale d'une architecture. Ces travaux sont le fruit d'une collaboration avec un doctorant du laboratoire Lab-STICC, Saeed Ur Rehman.

Perspective

La relaxation mémoire que nous avons proposée peut engendrer dans certains cas un surcoût matériel important dû à la présence d'éléments de mémorisations supplémentaires pour stocker les données conflictuelles. Par conséquent, si le concepteur n'exige pas la génération d'un réseau d'interconnexion particulier, il serait intéressant à court termes de définir dès le départ un seuil maximal pour le nombre de registres et de multiplexeurs associés. Si cette borne ne peut être respecter durant l'exploration de l'espace des solutions, alors le réseau cible devrait être modifié.

A moyen terme, l'objectif est d'offrir un espace d'exploration des solutions d'allocation mémoire plus large, permettant ainsi de réduire le coût architecturale de ces composants de brassage. Pour ce

faire il faut que les travaux en collaboration avec Saeed Ur Rehman soient finalisés et intégrés au sein du flot de placement mémoire généralisé.

A plus long terme, cette approche généralisée pourrait être embarquée au sein d'un circuit afin de calculer à la volée les placements mémoire. Ceci prend tout son sens dans le contexte des circuits multi-standards. En effet, si l'on se contente d'utiliser des ROMs mémorisant les séquences de contrôle des mémoires et du réseau, le nombre de ROMs nécessaires génèrera un surcoût architectural trop important (en termes de coût, de consommation…).

Bibliographie personnelle

REVUES INTERNATIONNALES

[BRI13] A. Briki, S. Ur Reehman, C. Chavet and P. Coussy, "--", *SOUMISSION A VENIR*

CONFERENCES INTERNATIONNALES

[BRI13a] A. Briki, C. Chavet and P. Coussy, "A Memory Mapping Approach for Network and Controller Optimization in Parallel Interleaver Architectures", *In Proceedings of the 23th ACM Great Lakes Symposium on VLSI (GLSVLSI) 2013, page XX-YY, Paris, France, may 2013*

[BRI13b] A. Briki, C. Chavet and P. Coussy, "Network and Controller Optimization in Parallel Interleaver Architectures, New Memory Mapping Approach", *In Proceedings of the IEEE Workshop on Signal Processing Systems (SiPS) 2013, page XX-YY, Tapei, Taiwan, oct. 2013 – SOUMISSION EN COURS DE REVIEW*

[BRI12a] A. Briki, C. Chavet, P. Coussy and E. Martin, "A Design Approach Dedicated to Network-Based and Conflict-Free Parallel Interleavers", *In Proceedings of the 22th ACM Great Lakes Symposium on VLSI (GLSVLSI) 2012, page XX, Salt lake City, USA, may 2012*

CONFERENCES NATIONNALES

[BRI13c] A. Briki, P. Coussy, C. Chavet, and E. Martin, "Placement de données en mémoire sans conflit pour l'optimisation du réseau d'interconnexion et du contrôleur des entrelaceurs parallèles", *In Colloque National du GRETSI 2013.*

[BRI12b] A. Briki, P. Coussy, C. Chavet, and E. Martin, "A Design Approach Dedicated to Pattern-Based and Conflict-Free Parallel Memory Systems", *In Colloque National du GDR SoC-SiP, Paris, France, june 2012.*

[BRI11] A. Briki, P. Coussy, C. Chavet, and E. Martin, "A Design Approach Dedicated to Pattern-Based and Conflict-Free Parallel Memory System", *In Colloque National du GDR SoC-SiP, Lyon, France, june 2011*

BIBLIOGRAPHIE

[AHA] "Primer: Reed-Solomon Error Correction Codes", *AHA Application Note*.

[ASG10] Rizwan Asghar and Dake Liu "Towards Radix-4, Parallel Interleaver Design to Support High-Throughput Turbo Decoding for Re-Configurability" *33rd IEEE SARNOFF Symposium 2010*, Princeton, NJ, USA.

[BAH74] L. R. Bahl, J. Cocke, F. Jelinek and J. L. R. Bahl, J. Cocke, F. Jelinek and J. Raviv, "Optimal decoding of linear codes for minimizing symbol error rate," *IEEE Trans. Inform. Theory*, vol. IT-20, no. 2, pp. 284–287, March 1974.

[BAT04] A. Batra et al., "Design of a multiband OFDM system for realistic UWB channel environments," *IEEE Trans. Microwave Theory Tech*., vol. 52, no. 9, pp. 2123–2138, Sept. 2004.

[BAZ97] M. S. Bazaraa, J. J. Jarvis, "Linear Programming and Network Flows". Chapter 8. *John Wiley and Sons*. 1997.

[BER93] C. Berrou, A. Glavieux and P. Thitimajshima, "Near Shannon limit error-correcting coding and decoding: turbo-codes" in *Proc. IEEE Int. Conf. on Communications (ICC 93)*, Geneva, Switzerland, pp. 1064–1070, 1993.

[BLA02] A. J. Blanksby and C. J. Howland, "A 690-mW 1-Gb/s 1024-b, rate-1/2 low-density paritycheck code decoder," *IEEE J. Solid-State Circuits*, vol. 37, no. 3, pp. 404–412, March 2002.

[BLA05] Blankenship, B. Classon, and V. Deai. "High-throughput turbo decoding techniques for 4G" *In Proc. Int. Conf. 3G Wireless and Beyond*. San Francisco, CA, page 137142, June 2005.

[BRU46] De Bruijn, N.G., "A Combinatorial Problem" *Koninklijke Nederlandse Akademie* v. Wetenschappen 49,758–764, 1946.

[CHA10] C. Chavet, P. Coussy, "A memory Mapping Approach for Parallel Interleaver design with multiples read and write accesses". *In Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)* 2010.

[CHA10a] C. Chavet, P. Coussy, P. Urard and E. Martin, "Static Address Generation Easing: a Design Methodology for Parallel Interleaver Architecture". In *proc. ICASSP 2010*.

[CHA07] C. Chavet, C. Andriamisaina, P. Coussy, E. Casseau, E.Juin, P. Urard, and E. Martin, "A Design Flow Dedicated to Multi-mode Architectures for DSP Applications", In Proceedings of the IEEE International Conference on Computer Aided Design (ICCAD) 2007, pages 604-611, San Jose, USA, november 2007

[CHE02] J. Chen and M. Fossorier. "Density evolution of two improved bp-based algorithms for LDPC decoding". *IEEE Communication letters*, March 2002.

[COL82] R. Cole, J. Hopcroft, " On edge coloring bipartite graphs", *SIAM Journal on Computing* 11 (1982) 540-546.

[DAV] http://www.ict-davinci-codes.eu/

[DVBS08] *ETSI EN 302-583 V1.1.1, (2008).* "Digital Video Broadcasting (DVB); Framing structure, channel coding and modulation for satellite services to handheld devices (SH) below 3 GHz". March. 2008.

[DVB08] *DVB DocumentA122.* "Frame structure channel coding and modulation for the second generation digital terrestrial television broadcasting system (DVB-T2)," 2008.

[FAN06] L. Fanucci, P. Ciao and G. Colavolpe, "VLSI design of a fully-parallel high-throughput decoder for turbo Gallager codes," *IEICE Trans. Fund. Electronics, Communications and Computer Sci*., vol. E89-A, no. 7, pp. 1976–1986, July 2006.

[FOS99] M.P.C Fossorier, M. Mihaljevic, and H. Imai. "Reduced complexity iterative decoding of low-density parity check codes based on belief propagation". *IEEE Transactions on communications*, 47:673–680, May 1999.

[GAB76] H.N. Gabow, "Using Euler partitions to edge color bipartite multigraphs", *International Journal of Computer and Information Sciences 5* (1976) 345-355.

[GIU02] A. Giulietti, L. v. d. Perre, and A. Strum, "Parallel turbo coding interleavers: Avoiding collisions in accesses to storage elements," *Electron.Lett*., vol. 38, no. 5, pp. 232–234, Feb. 2002.

[GNA03] D.Gnaedig, E.Boutillon, M.Jezequel, V.C.Gaudet, and P.G.Gulak, "On multiple slice turbo codes," in *Proc. 3rd Int. Symp. Turbo Codes, Related Topics*, Brest, 2003, pp. 343–346.

[GOL61] M. J. E. Golay, "Complementary series", *IRE Trans. on Info. Theory*, vol. IT-7, pp. 82-87, April 1961.

[GRO03] Jonathan L. Gross, Jay Yellen, "Handbook of Graph Theory". *CRC Press*. 2003.

[HAM50] R.R. W. Hamming, "The Bell System Technical Journal," *Bell Syst. Tech. J*., vol. XXVI, no. 2, pp. 147–160, Apr. 1950.

[HAR06] D. Hartvigsen, "Finding maximum square-free 2-matcing in bipartite graphs". *J. combin. Theory*, Ser. B 96 (2006) 693-705.

[HSP04] 3GPP, "Technical specification group radio access network;multiplexing and channel coding (FDD)" *(25.212 V5.9.0).*June 2004.

[HSP08] 3GPP, "Technical Specification Group Radio Access Network; Multiplexing and Channel Coding (FDD)" *Tech. Spec. 25.212 V8.4.0* Dec. 2008.

[JOH10] Sarah J. Johnson, "Iterative Error Correction Turbo, Low-Density Parity-Check and Repeat–Accumulate Codes" *Cambridge University Press* 2010

[KAI05] Kaiser, Ed. "UWB Communications Systems: A Comprehensive Overview" *EURASIP Series on Signal Processing and Communications, Hindawi Publishing*, New York, 2005.

[KEY01] P. Keyngnaert, B. Demoen, B. De Sutter, B. De Sus, and K. De Bosschere. "Conflict Graph Based Allocation of Static Objects to Memory Banks" *Informal proceedings of the First workshop on Semantic, Program Analysis, and Computing Environments*, pages 131–142, September 2001.

[KIE03] Kienle, F., Thul, M. J., and When, N., 2003. "Implementation Issues of Scalable LDPC-Decoders". *in Proceeding of 3rd International Symposium on Turbo Codes and Related Topics, Brest, France,* 291-294.

[KON16] D. König, "Graphok és alkalmazásuk a determinánsok és a halmazok elméletére". *Mathematikai és Természettudományi Értzsitö* 34 (1916) 101-119.

[KOZ92] D.C.Kozen, "The Design and Analysis of Algorithms", *Springer Verlag, USA*, 1992.

[KWA02] J. Kwak and K. Lee, "Design of dividable interleaver for parallel decoding in turbo codes," *Electron. Lett*., vol. 38, no. 22, pp. 1362–1364, 2002.

[LEE08] J.-Y. Lee and H.-J. Ryu, "A 1-Gb/s flexible LDPC decoder supporting multiple code rates and block lengths," *IEEE Trans. Consumer Electronics*, vol. 54, no. 2, pp. 417–424, May 2008.

[LIN04] Shu Lin and Daniel J. Costello, Jr., "Error control Coding" *Pearson Education*, Inc 2004

[LIN10] Jing-ling, "Parallel Interleavers Through Optimized Memory Address Remapping" *IEEE Trans. VLSI Systems* vol. 18, no.6, pp.978-987, June. 2010.

[LTE08] "Technical Specification Group Radio Access Network; Evolved Universal Terrestrial Radio Access; Multiplexing and Channel Coding (Release 8)", *3GPP Std. TS 36.212*, Dec. 2008.

[MAN03] M. Mansour and N. Shanbhag, "High-throughput LDPC decoders," *IEEE Trans. VLSI Syst*., vol. 11, no. 6, pp. 976–996, Dec 2003.

[MAS07] G. Masera, F. Quaglio and F. Vacca, "Implementation of a flexible LDPC decoder," *IEEE Trans. Circuits and Systems – II: Express Briefs*, vol. 54, no. 6, pp. 542–546, June 2007

[MOU07] H. Moussa, O. Muller, A. Baghdadi, and M. Jezequel, "Butterfly and Benes-based on-chip communication networks for multiprocessor turbo decoding," in *Proc. of the conference on Design, Automation and Test in Europe*, pp. 654-659, April 2007.

[MOU08] H. Moussa, A. Baghdadi, and M. Jezequel, "Binary de Bruijn interconnection network for a flexible LDPC/turbo decoder," in *Proc. IEEE Int. Symp. Circuits Syst.*, 2008, pp. 97–100.

[MOU08] H.Moussa, A.Baghdadi, M.Jezequel."Binary de Bruijn on-chip netwok for a flexible multiprocessor LDPC decoder".*45th ACM/IEEE DAC, p.429-434*, 2008.

[NAG04] V. Nagarajan, N. Jayakumar, S. Khatri and G. Milenkovic, "High-throughput VLSI implementations of iterative decoders and related code construction problems," in Proc. *IEEE Global Telecommunications Conf. (GLOBECOM 2004)*, vol. 1, pp. 361–365, 2004.

[NEE05] C. Neeb, M. Thul, and N. Wehn, "Network-on-chip-centric approach to interleaving in high throughput channel decoders," in *Proc. IEEE International Symposium on Circuits and Systems (ISCAS),* May 2005, pp. 1766 – 1769 Vol. 2.

[ORE67] Ore, "The Four Color Problem" *Academic Press*, New York, 1967.

[PEA88] J.Pearl, "Probabilistic Reasoning in Intelligent Systems: Networks of Plausible reference", *Morgan Kaufmann*, 1988.

[QUA06] F.Quaglio, F.Vacca, C.Castellano, A.Tarable, M.G.Asera. "Interconnection Framework for High-Throughput, Flexible LDPC Decoders". *In proceeding Design Automation and Test in Europe Conference and Exhibition*, 2006.

[SAN10] Awais Sani, Philippe Coussy, Cyrille Chavet, Eric Martin, "Design of parallel LDPC interleaver architecture: A bipartite edge coloring approach", *ICECS 2010*, 466-469.

[SAN11] Awais Sani, Philippe Coussy, Cyrille Chavet, Eric Martin "An approach based on edge coloring of tripartite graph for designing parallel LDPC interleaver architecture" *ISCAS 2011*, 1720-1723.

[SAN11a] Awais Sani, Philippe Coussy, Cyrille Chavet, Eric Martin ”A methodology based on Transportation problem modeling for designing parallel interleaver architectures” *ICASSP 2011*, 1613-1616.

[SCH98] A.Schrijver, “Bipartite edge coloring in O(Δ*m*) time”, *SIAM Journal on Computing*, vol.28, pp.841–846, 1998.

[SHA48] C.E. Shannon, A mathematical theoryof communication, *Bell System Tech*. J. 27 (July and October 1948) 379–423, 623–656.

[SIR08] W. P. Siriwongpairat, K. J. Ray Liu “Ultra-Wideband Communications Systems” *Joh Wiley & Sons, Ins., Publication,* 2008.

[SUN05] J. Sun and O. Y. Takeshita, “Interleavers for turbo codes using permutation polynomials over integer rings ,” *IEEE Trans. Inf. Theory*, vol. 51, no. 1, pp. 101–119, Jan. 2005.

[TAK06] O. Y. Takeshita, “On maximum contention-free interleavers and permutation polynomials over integer rings,” *IEEE Trans. Inf. Theory*, vol. 52, no. 3, pp. 1249–1253, Mar. 2006.

[TAR04] A. Tarable, S. Benedetto, and G. Montorsi, “Mapping interleaving laws to parallel turbo and LDPC decoder architectures”, *IEEE Trans.Inf.Theory*, vol. 50, no.9, pp.2002-2009, Sep. 2004.

[TAR05] A. Tarable and S. Benedetto, “Further results on mapping functions,” *in Proc. Inform. Th. Workshop 2005 (ITW2005)*, pp. 221-225.

[THE05] Theocharides, T., Link, G., Vijaykrishnan, N., and Irwin, M. J. “Implementing LDPC Decoding on a Network-on-Chip”. I*n Proc. of the Int. Conf.* VLSI Design, 2005, 134-137.

[THU02] M. Thul, N. Wehn, and L. Rao, “Enabling high-speed turbo decoding through concurrent interleaving,” in *Proc. IEEE International Symposium on Circuits and Systems (ISCAS)*, vol. 1, 2002, pp. 897–900.

[THU02a] M. I. Thul, F. Gilbert. and N. Wehn. “Optimized Concurrent Interleaving for High-speed Turbo-Decoding”. In *Proc. 9rh IEEE International Conference on Electronics, Circuits and Systems - ICECS* 2002, Dubrovnik, Croatia, Sept. 2002.

[THU03] M. Thul, F. Gilbert, and N. Wehn, “Concurrent interleaving architectures for high-throughput channel coding,” in *Proc. IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP),* 2003, pp. 613–616 vol.2.

[WIF08] *IEEE P802.11n/D5.02, Part 11*. “ Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications: Enhancements for Higher Throughput” , July 2008.

[WIM06] *IEEE P802.16e, Part 16*. “ Air Interface for Fixed and Mobile Broadband Wireless Access Systems,” Amendment 2: Physical and Medium Access Control Layers for Combined Fixed and Mobile Operation in Licensed Bands, and Corrigendum 1, Feb. 2006.

[WON10] C.-C.Wong and H.-C. Chang, “Reconfigurable turbo decoder with parallel architecture for 3GPP LTE system,” *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 57, no. 7, pp. 566–570, Jul. 2010.

[WOO00] J. P. Woodard and L. Hanzo, “Comparative study of turbo decoding techniques: An overview,” *IEEE Trans. Veh. Tech*., vol. 49, no. 6, pp. 2208–2233, Nov. 2000.

[ZHA01] T. Zhang and K. K. Parhi. “Joint code and decoder design for implementation-oriented (3;k)-regular LDPC codes”. *in Proc. Asilomar Conference on Signals, Systems and Computers*, 2:1232_1236, Nov.2001.

[ZHA04] Y. Zhang and K. Parhi, “Parallel turbo decoding,” *in Proceedings of the ISCAS ’04.*, vol. 2, 23-26 May 2004, pp. II–509–12Vol.2.

[ZHO07] L. Zhou, C.Wakayama and C.-J. R. Shi, “CASCADE: a standard super-cell design methodology with congestion-driven placement for three-dimensional interconnect-heavy very large scale integrated circuits,” *IEEE Trans. Computer-Aided Design*, no. 7, July 2007