Conception des blocs réutilisables. Réflexion sur la méthodologie

B. Laurent

THESE

présentée par:

Bernard LAURENT

pour obtenir le grade de **DOCTEUR**

de l'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

(Arrêté ministériel du 30 mars 1992)

spécialité: **Microélectronique**

Conception des Blocs Réutilisables Réflexion sur la Méthodologie

Date de la soutenance: **18 juin 1999**

Composition du Jury:

Pierre	Gentil	Président
Michel	Robert	Rapporteur
Anne	Mignotte	Rapporteur
Jean-François	Agaesse	Examinateur
Ahmed-Amine	Jerraya	Directeur de Thèse

Thèse préparée au sein du Laboratoire TIMA
46, Avenue Félix Viallet, 38031 Grenoble

Remerciements

Je remercie Pierre Gentil, Professeur et Directeur de l'école doctorale de micro-électronique, qui m'a fait l'honneur de bien vouloir présider ce jury. Je remercie mes rapporteurs Michel Robert, Professeur de l'université de Montpellier II au sein du laboratoire LIRMM, et Anne Mignotte, Maître de conférence à l'école centrale de Lyon au sein du laboratoire LIP.

Je remercie Bernard Courtois, Directeur de Recherche au CNRS et Directeur du TIMA, d'avoir bien voulu m'accueillir dans son laboratoire. Je remercie Ahmed-Amine Jerraya, Directeur de Recherche au CNRS, d'avoir dirigé ce travail de thèse. Je remercie également Jean-François Agaësse pour m'avoir accueilli dans la société Thomson-CSF semi-conducteurs spécifiques et pour avoir fait partie de mon jury de thèse.

Je remercie les membres du CSI, du TIMA et de TCS pour les moments partagés au cours de ces trois années universitaires, en particulier Mohammed Belrhiti, Gilles Bosco et Tarik Kannat pour avoir partagé une partie de leurs recherches. Je remercie Bobby, Christian, Sid-Ahmed, Ali, Adel, Toufik, Jean-Pierre, Marie-Claude, Raphaël, Xavier, Hichem, Helena, Abdelkader, Laurent, Jean-Paul, Catherine, Denis, Sophie, Sylvie, Hervé et Christophe.

Résumé

L'évolution des technologies, les exigences de productivité, l'accroissement de la complexité des circuits intégrés ont contribué à l'émergence des composants virtuels (IPs), ainsi qu'au développement de logiciels d'aide à la conception de circuits intégrés. L'utilisation de l'abstraction et des composants déjà conçus sont les clés de ces défis.

L'objet de cette thèse est le parcours des principaux niveaux d'abstraction de la synthèse matérielle, la synthèse logique, RTL et comportementale, en dégageant pour chacun d'entre eux les contraintes de conception qui vont devenir les critères de sélection d'un bloc réutilisable. Il ne reste plus qu'à concevoir un éventail de blocs dans une approche de réutilisation: les blocs doivent être facilement sélectionnables, puis paramétrables, et enfin intégrables dans un circuit plus important. La conception des blocs comportementaux, appliquée au codage correcteur d'erreur, nous amène à réfléchir sur les méthodologies de conception et de réutilisation des composants virtuels.

mots clés : conception, composants virtuels, IP, décomposition technologique, arithmétique, codage correcteur d'erreur.

Abstract

Technological progress, designers' productivity expectations, exponential increase of design complexity, have lead to the emergence of virtual components (IPs), as well as the development of software for integrated circuits design automation. The use of abstraction and components already designed are the keys of these challenges.

The aim of this thesis is the study of the main abstraction levels of hardware synthesis, logic, RT and behavioral synthesis, in order to deduce, for each of them, the optimization constraints that will become the criteria of selection for a reuse block. Then, one has to design an array of blocks in a reuse approach: the blocks must be easily selected, parameterized, and then integrated in a complete circuit. The design of behavioral blocks, applied to the error correcting codes, gives us food of thought about the design and reuse methodologies of virtual components.

Table des Matières

Liste des Figures

Liste des Tableaux

Chapitre 1

Introduction Générale

1.1 Cadre général

L'évolution des technologies, les exigences de productivité, l'accroissement de la complexité des circuits intégrés ont contribué à l'émergence des composants virtuels (IPs) ainsi qu'au développement de logiciels d'aide à la conception de circuits intégrés. Ces outils et composants permettent d'augmenter la productivité des concepteurs en les déchargeant de tâches fastidieuses et longues, tout en garantissant un résultat final de bonne qualité et exempt de défauts.

Actuellement, la complexité des circuits ne permet pas de les concevoir sans faire appel à des méthodes de réutilisation ou à l'utilisation massive d'outils automatiques. L'utilisation de l'abstraction et des composants déjà conçus sont les clés de ce défi.

1.2 Motivations

Il faut donc concevoir des composants en vue de leur réutilisation dans plusieurs niveaux d'abstraction. L'objet de cette thèse est le parcours des principaux niveaux d'abstraction de la synthèse matérielle en dégageant pour chacun d'entre eux les contraintes de conception qui vont devenir les critères de sélection d'un bloc réutilisable. Une fois ces contraintes déterminées, il ne reste plus qu'à concevoir un éventail de blocs dans une approche de réutilisation: les blocs doivent être facilement sélectionnables, puis paramétrables, et enfin intégrables dans un circuit plus important. Une première étude consiste donc à:

fournir un ensemble de blocs réutilisables pouvant être aisément sélectionnés, paramétrés et intégrés.

L'utilisation des ces blocs peut se faire à plusieurs niveaux de la synthèse matérielle: dans la synthèse comportementale en utilisant une fonctionnalité complexe comme pas de calcul, dans la synthèse RTL en produisant des macroblocs arithmétiques satisfaisants à toutes les contraintes de l'optimisation, et enfin dans la synthèse logique en facilitant l'utilisation massive des cellules standards des bibliothèques cibles. Les contraintes changent à chacun des niveaux de description: il s'agit du débit, de la surface mais aussi des contraintes technologiques et temporelles ou encore de la puissance dissipée.

Décrire des blocs réutilisables n'est pas tout. Encore faut-il qu'ils puissent être suffisamment fiables. La conception des blocs comportementaux, c'est-à-dire ayant une latence supérieure au cycle d'horloge, nous amène à réfléchir sur les méthodologies de conception et de réutilisation. Il faut:

définir des méthodes de conception et de réutilisation afin de construire et utiliser les composants virtuels

1.3 Plan de lecture

La thèse se compose comme suit. L'ensemble de la synthèse matérielle est présentée en *chapitre 2*: le découpage en couches d'abstaction, les critères d'optimisation ainsi que les blocs réutilisables.

Contrairement aux autres chapitres consacrés à l'élaboration d'une bibliothèque de blocs, le *chapitre 3* propose d'améliorer l'outil de décomposition technologique afin de mieux utiliser les ressources disponibles. En particulier, nous nous efforcerons de faciliter l'utilisation des portes complexes des bibliothèques de cellules standards afin de diminuer la puissance dissipée.

Le *chapitre 4* se consacre à la conception d'un macrogénérateur de blocs arithmétiques. Les contraintes du concepteur sont de deux types: l'environnement temporel et la cible technologique. Aussi avons nous construit un générateur d'additionneurs et de multiplieurs répondant aux contraintes temporelles et technologiques afin de produire des blocs arithmétiques optimisés pour chaque jeu de contraintes.

Le *chapitre 5* s'attache à construire une bibliothèque de composants virtuels (IPs) sur la détection et la correction d'erreur de transmission. Les codes de Reed-Solomon et de Viterbi sont étudiés en détail ainsi que leurs multiples réalisations matérielles. Les contraintes de surface et de débit nous amènent à proposer plusieurs architectures afin de couvrir toutes les exigences de performances.

Le *chapitre 6* enfin apporte des éléments de réflexion sur les aspects méthodologiques: les principales méthodes de conception et de réutilisation sont revues et nous apportons une méthode assurant une plus grande sûreté et rapidité de conception, que nous avons bien entendu appliquée à l'ensemble des blocs de haut-niveau.

Chapitre 2

Les synthèses de circuits intégrés

2.1 Exigences de productivité

Les objectifs majeurs de la conception VLSI sont l'amélioration de la qualité et de la productivité. La raison en est simple: d'une part le budget alloué à la conception de circuit est fixe (10 personnes sur 18 mois par exemple). D'autre part, la complexité croît de façon exponentielle depuis 1984 d'une centaine de milliers de transistors à quelques millions de transistors aujourd'hui. Cette complexité devrait atteindre 40 millions de transistors dès l'an 2010. Si nos méthode de conception sont suffisantes pour le moment, il faudrait faire face à un facteur de 20 à 40 pour combler le progrès technologique. La solution à ce défi est double [1]:

- Concevoir un circuit intégré à un niveau d'abstraction plus élevé. La synthèse comportementale permet, pour un même volume de code et de temps de conception, de réaliser un circuit beaucoup plus important puisque les détails architecturaux sont implantés automatiquement.
- Utiliser les composants et sous-systèmes existants au lieu de les concevoir. Ceci permet d'intégrer un composant en un temps constant, indépendamment de sa complexité, au lieu de le réaliser en un temps dépendant de sa complexité. La plupart des circuits existants n'utilisent pas ces composants par manque de méthodologie.

2.2 L'abstraction

L'abstraction, d'un composant par exemple, est une description succincte qui supprime les détails d'implantation, inutiles pour en comprendre la fonctionnalité. Le logiciel a plusieurs niveaux d'abstraction consécutifs: au plus bas se trouve le langage machine, puis le langage assembleur. Encore plus haut se place le langage C qui libère le programmeur, par exemple, de la gestion des registres internes aux processeurs. Dans certains langages comme ADA, le module de spécification peut servir

de niveau d'abstraction supplémentaire.

De cet exemple, C.W.Krueger identifie dans chaque abstraction deux niveaux [2]: le niveau succinct le plus élevé, ou **spécification**, et le niveau détaillé le plus bas, ou **réalisation**. Lorsque plusieurs couches d'abstraction sont employées, comme dans l'exemple logiciel précédent, la spécification d'un niveau aval est la réalisation du niveau amont. L'illustration 2.1 montre comment la représentation intermédiaire 2 est incluse dans deux couches d'abstractions: comme spécification de la première abstraction et comme réalisation de la seconde. Elle est donc de taille plus importante que sa spécification mais aussi plus petite que sa réalisation.

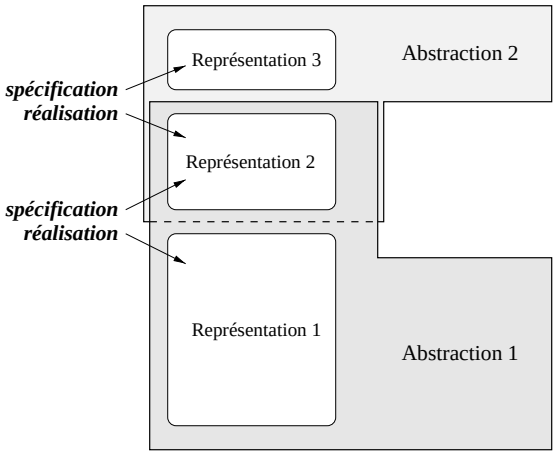

Figure 2.1: Les couches d'abstraction.

On distingue dans chaque abstraction trois parties conditionnant l'obtention d'une réalisation: la partie cachée et la partie visible, elle-même divisée en partie fixe et variable. La **partie cachée** consiste en tous les détails d'implantation qui ne sont pas visibles dans la spécification. Cela peut inclure, par exemple, l'algorithme utilisé. La **partie fixée** (visible) regroupe les caractéristiques invariantes de la réalisation. Elle limite la généricité des réalisations possibles. Enfin, la **partie variable** (visible) représente les caractéristiques que le concepteur peut faire évoluer afin de paramétrer au mieux son composant. Le choix d'appartenance d'une caractéristique n'est pas intrinsèque à l'objet réalisé. C'est plutôt un choix du concepteur de rendre visible et générique ses réalisations en augmentant la partie variable. Il est en effet plus simple de rendre cachées ou fixes les caractéristiques d'un composant, mais celui-ci devient moins attractif puisqu'il sera très spécifique. Nous nous proposons donc de définir quelles sont les couches d'abstraction contenues dans la synthèse des circuits intégrés.

2.3 Les étapes de la synthèse

2.3.1 Vue d'ensemble

Le flot de conception d'un circuit consiste à réaliser au niveau physique une description algorithmique. Alors que pour les circuits simples utilisant un nombre limité de portes, l'attention du concepteur se portait sur l'optimisation au niveau silicium, les circuits actuels sont si complexes qu'il a bien fallu développer un ensemble de couches d'abstraction afin de diviser le flot de synthèse en sous-problèmes simplifiés. Le processus de synthèse, présentée en figure 2.2, apparaît alors comme une cascade de niveaux d'abstraction consécutifs.

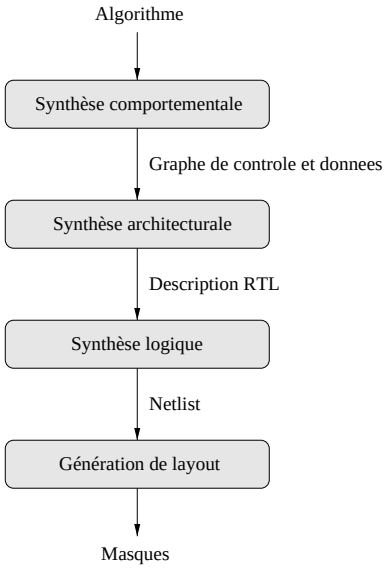

Figure 2.2: Le processus de synthèse.

La description d'entrée est un algorithme écrit dans un langage de haut niveau tel que VHDL (Norme IEEE 1076) ou Verilog. La synthèse comportementale extrait une partie contrôle et opérative en allouant les opérateurs et ordonnançant les opérations suivant les contraintes du concepteur. La synthèse architecturale associe les opérateurs aux modules de librairie et les variables aux points mémoires pour obtenir une représentation en transfert de registres. Alors seulement, la synthèse logique optimise cette représentation pour fournir un réseau de portes explicite pour les parties combinatoires et séquentielles. Enfin, le circuit est placé et routé pour aboutir à un ensemble de masques prêt à être fabriqué.

2.3.2 Synthèse comportementale

Le circuit est spécifié en terme de pas de calcul, séparés par des points de synchronisation ou des lectures/écritures des entrées/sorties. L'objectif de la **synthèse comportementale** est de découper ces pas en ensembles de cycles d'horloge pour fournir une architecture synchrone. La synthèse comportementale agit comme un compilateur qui décompose une spécification de haut niveau en une architecture. Une pléthore d'outils ont été publiés dans la littérature. Citons les outils HYPER [91], de Lis et Gajski [92], HAL [93], SPLICER [94], CAMAD [95], HERCULES [96]. Aucun de ces outils, il est vrai. a été reconnu comme l'approche universelle pour tous les domaines d'application et toutes les architectures cibles. Les différences correspondent au choix qui doivent être faits, comme:

- Le domaine d'application, parmi lesquels les opérateurs de DSP, les contrôleurs ou interfaces, et donc le style de l'architecture produite.
- Les modèles intermédiaires utilisés pour représenter le circuit. Ils sont majoritairement basés sur des graphes de flot et des machines d'états finis, parmi lesquels: le graphe de flot de donnée, le graphe de flot de contrôle, le graphe de flot de contrôle et de donnée utilisé dans l'outil *BC* [97], et les machines d'états munies d'un modèle de chemin de données ou de co-processeurs.
- La flexibilité du processus de synthèse qui propose des étapes contrôlables par le concepteur et d'autres pas.
- Les descriptions d'entrée et de sortie.
- Les algorithmes utilisés pour chaque étape du flot de synthèse qui peuvent varier d'un outil à l'autre. Citons, par exemple, les ordonnancements au plus tôt, au plus tard, ou dirigés par les forces. On peut également réaliser plusieurs tâches en parallèle ou séparément.

Toutefois, on peut dire qu'une fois le code source débarassé de tous les détails relatifs au langage de description, la synthèse comportementale peut se décomposer en trois étapes majeures, comme ceci est présenté dans [98], réalisé dans l'outil de synthèse *AMICAL* développé au laboratoire TIMA [99] et présenté dans la figure 2.3:

- L'**ordonnancement** partitionne la description comportementale en sous-graphes exécutables durant un cycle d'horloge. Le compromis entre la latence et le parallélisme est réalisé sous contraintes de ressources et de temps [3]. Puisque la surface du circuit est directement fonction du nombre de ressources utilisées, l'étape d'ordonnancement conditionne les performances de l'architecture produite.

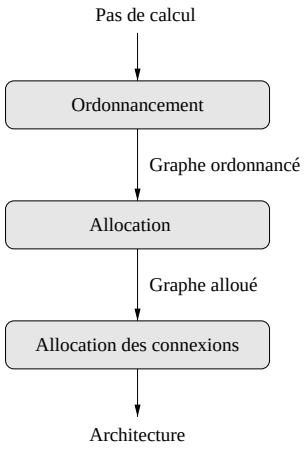

Figure 2.3: La synthèse comportementale.

- L'**allocation** associe aux opérations précédentes les opérateurs disponibles en bibliothèque. L'objectif est non seulement de minimiser le nombre d'opérateurs, mais aussi de choisir intelligemment les points mémoires. Ceci permet de limiter la connectique provoquée par le partage trop fréquent des éléments séquentiels.
- Enfin, l'**allocation des connexions** détemine les ressources nécessaires à assurer la communication entre les unités du chemin de données. Deux solutions sont proposées: une architecture à base de multiplexeurs ou de bus. L'architecture finale est alors générée, le plus souvent comme un contrôleur séquençant les pas de contrôle et activant les composants à chacun d'entre eux [4].

2.3.3 Synthèse RTL

Une fois l'ordonnancement effectué, il ne reste plus qu'à optimiser les opérations de chaque période d'horloge. Toutefois, on ditingue une étape intermédiaire entre la synthèse comportementale et l'optimisation: la **synthèse RTL** ou transfert de registres. Celle-ci transforme un circuit spécifié pour chaque cycle d'horloge en un ensemble d'équations booléennes en effectuant, elle aussi, une allocation de ressources. Comme ceci est réalisé dans l'outil de synthèse RTL ASYL+ [30], et illustré dans la figure 2.4, on distingue l'élaboration et la macrogénération:

- L'**élaboration** consiste à compiler le code source en un modèle interne où sont spécifié l'ensemble des registres, des opérateurs et les transferts entre registres et opérateurs. Ce modèle dégage les parties séquentielles et asynchrones au vu de la syntaxe RTL utilisée.
- La **macrogénération** transforme les machines et opérateurs en équations: les opérateurs, arithmétiques par exemple, sont transformés en équations contenues

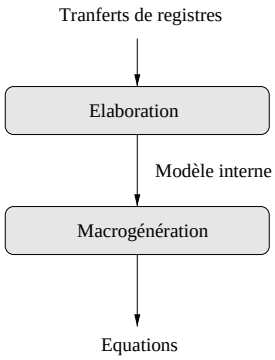

Figure 2.4: La synthèse RTL.

dans les éléments de bibliothèque et les états des machines d'états finis sont encodés. La structure d'une telle génération se décide en fonction des contraintes du concepteur, puisque l'optimisation logique ne pourra changer la structure de la macro.

2.3.4 Synthèse logique

La **synthèse logique** génère un ensemble de portes, ou *netlist*, à partir d'une description logique sous forme d'équations booléennes. On distingue souvent synthèse 2-couches et synthèse multi-couches.

- La **synthèse 2-couches** indique qu'un signal d'entrée traverse au plus dans le circuit 2 niveaux de portes logiques pour arriver à la sortie pourvu que les signaux soient disponibles sous forme directe ou complémentée. L'optimisation d'une telle représentation est la **minimisation**, ou décomposition de la fonction en une somme réduite de monômes [5].

- La **synthèse multi-couches** permet un nombre quelconque de portes sur le chemin critique et se déroule comme la suite logique de la synthèse 2-couches: une étape de **factorisation** met en commun une partie des monômes précédents afin de minimiser le nombre de littéraux (variables). La surface en portes du circuit est en effet proportionnelle au nombre de littéraux [6].

Une fois les équations optimisées, celles-ci sont **décomposées** sur la bibliothèque de cellules dont le concepteur dispose. Cette étape de recouvrement ne modifie pas la structure des équations, mais choisit un ensemble optimal de portes pour minimiser le ou les critères d'optimisation. L'ensemble de la synthèse logique est représenté en figure 2.5: seule la phase de décomposition dépend de la technologie. La minimisation et factorisation sont évaluées par la complexité des équations booléennes, en termes de nombre de monômes et de littéraux.

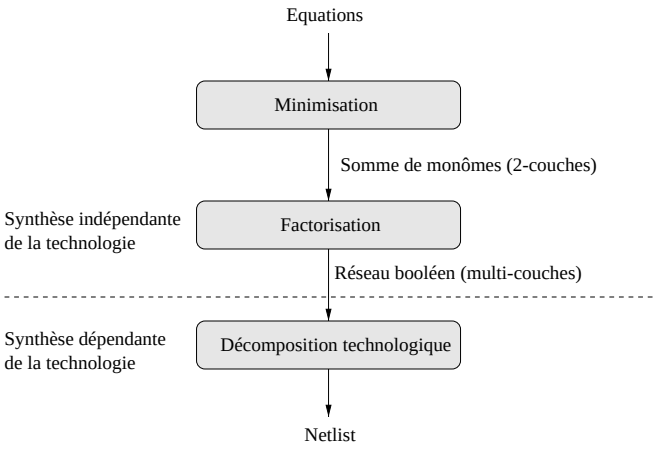

Figure 2.5: La synthèse logique.

2.4 Les blocs réutilisables

2.4.1 Utilisation des macroblocs

Nous entendons par macrobloc un élément d'une bibliothèque dont le concepteur dispose et qu'il peut directement inférer ou instantier sans avoir à le concevoir. Utiliser des macroblocs consiste à choisir une approche bibliothèque, c'est-à-dire à parcourir tout au long du processus de synthèse cette bibliothèque pour en sélectionner les éléments dont il a besoin. Par opposition, le processus de synthèse pourrait recréer chacun des blocs lorsqu'il en a besoin, et les optimiser en fonction des contraintes. Ici, nous sélectionnons plutôt que nous concevons.

Force est de constater que les méthodes de réutilisation ne sont pas assez répandues. Les concepteurs préfèreraient réutiliser plutôt que de concevoir mais leur capacité à réutiliser n'augmente pas avec l'expérience. Frakes et Fox [7] concluent que le manque de formation et de méthode est à l'origine de ce retard.

2.4.2 Les macroblocs dans la synthèse

Les blocs se répartissent entre les différentes couches d'abstraction comme le fait le processus de synthèse. A chaque étape correspond un type de macrobloc. Comme présenté précédemment, la synthèse de circuits intégrés réalise une description algorithmique en un réseau de portes. Au cours du processus, la contrainte temporelle se précise: la synthèse comportementale utilise comme unité de temps le pas de contrôle pendant lequel se déroule une partie de l'algorithme. La synthèse RTL suppose que l'ensemble des opérations combinatoires spécifiées se déroulent dans le même cycle d'horloge. Enfin, la synthèse logique cherche à minimiser ce cycle en proposant une

implémentation matérielle la plus rapide possible.

Ce critère temporel est aussi utilisé pour classifier les blocs puisque ceux-ci doivent être utilisés dans l'une des étapes de la synthèse. La classification est la suivante: lorsque la latence du bloc est supérieure au cycle d'horloge, c'est-à-dire lorsqu'elle utilise plusieurs cycles d'horloge, celui-ci est un **bloc comportemental**, parcequ'il devra être utilisé comme une partie d'un pas de contrôle. Lorsque la latence du bloc peut-être moindre que la période d'horloge et peut-être inférée par le compilateur RTL, celui-ci est un **bloc RTL**. Les additionneurs et multiplieurs en sont les plus populaires: ils sont inférés par les opérateurs $+, *$ du VHDL. Enfin, lorsque le délai d'un bloc est inférieur à une période d'horloge mais qu'il n'existe pas de contrepartie VHDL ou Verilog, il s'agit d'un **bloc logique**. C'est le cas de toutes les cellules d'une bibliothèque d'ASIC.

Le processus de synthèse utilise la bibliothèque qui correspond au niveau d'abstraction dans lequel celui-ci se trouve. Nous avons constitué trois niveaux d'abstraction. Aussi on utilise trois types de bibliothèques, comme illustré sur la figure 2.6: chacune des bibliothèques est utilisée dans l'étape de la synthèse correspondant à la latence des blocs réutilisables qu'elle contient. L'outil utilise donc un bloc comportemental pour réaliser un pas de calcul en synthèse comportementale, un bloc RTL pour réaliser un opérateur arithmétique dans la phase de macrogénération, et un bloc logique pour réaliser une expression booléenne en synthèse logique. Une fois inféré, le bloc subira la synthèse des étapes en aval comme si celui-ci avait été conçu.

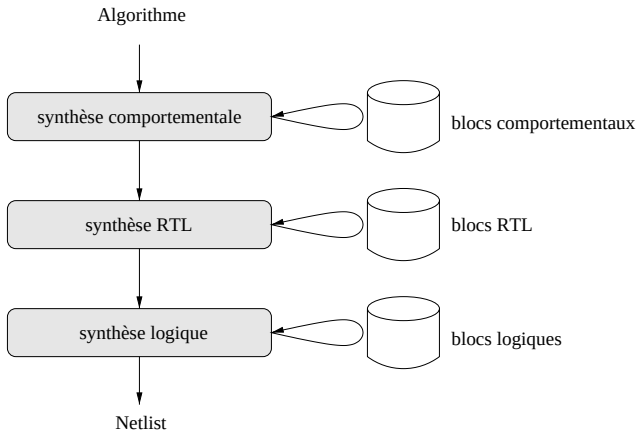

Figure 2.6: Bibliothèques de macroblocs.

2.5 Critères d'optimisation des circuits combinatoires

Les principaux critères d'optimisation des circuits intégrés sont: la surface, la fréquence d'horloge, la latence et la puissance dissipée (parfois la fiabilité).

2.5.1 Surface

La **surface** des circuits combinatoires se décompose en surface active et surface des interconnexions. La surface active repésente la surface des portes combinatoires de la bibliothèque cible effectuant une opération booléenne. La surface des interconnexions ne représente, pour un des processus technologiques utilisés dans le placement et routage de nos macros, que 15% environ de la surface totale (il s'agit effectivement d'un résultat expérimental constaté sur la technologie ST 0.35 μm). On en déduit que la surface active, constante, est une bonne estimation de la surface des circuits combinatoires, et que le choix d'une réalisation n'influant que peu sur la surface des interconnexions correspondantes, on peut les négliger dans le processus de sélection des portes logiques. Il est bien sûr indispensable de s'assurer que le circuit généré soit complètement routable.

Deux caractéristiques de la surface (active) sont importantes dans le processus de synthèse: premièrement, celle-ci est additive, et deuxièmement la surface d'une cellule est indépendante du choix des autres cellules. L'optimisation de la surface consiste donc à diminuer la surface de toutes les régions des circuits combinatoires sans préférence. De plus, le choix d'une cellule peut se faire indépendamment des choix déjà réalisés. Cette liberté assurera la simplicité et l'optimum du processus de sélection.

2.5.2 Fréquence d'horloge

La **fréquence d'horloge** est l'inverse du plus long chemin contenu entre les ports ou les éléments séquentiels. La durée du chemin critique est la somme du temps de traversée des cellules contenues dans ce chemin. Or, le temps de traversée d'une cellule peut être très bien estimé par la somme d'un temps intrinsèque, α, correspondant au temps de basculement des transistors MOS et du chargement ou déchargement de la capacité intrinsèque de la cellule, et d'un temps extrinsèque, β, correspondant au temps de chargement ou déchargement des capacités attaquées (cellules en amont C_{cell} et interconnexions C_{inter}) [8] selon:

$$t = \alpha + \beta(C_{inter} + C_{cell}). \tag{2.1}$$

Pour les technologies submicroniques toutefois, c'est-à-dire pour lesquelles la dimension des transistors est inférieure au micromètre, le temps de montée des tensions

(*slew rate*) ainsi que le délai des interconnexions interviennent, voire dominent dans certains cas. Afin de garder le processus de sélection optimal, nous garderons cette équation avec des valeurs statistiques du délai des interconnexions et des coefficients β et C_{inter} dépendant de la charge attaquée, de la sortance et de la surface du circuit combinatoire.

Deux caractéristiques du chemin critique sont importantes: premièrement celui-ci est le maximum de tous les chemins (et non plus la somme), et deuxièmement le délai d'une cellule dépend de la charge. C'est pourquoi le processus de sélection est plus complexe dans le cas de la fréquence d'horloge, mais peut toujours être optimal.

2.5.3 Latence

La **latence** d'un circuit correspond au temps écoulé entre l'entrée des données et leur sortie. Pour un circuit purement combinatoire, il s'agit du chemin critique. Mais pour un circuit séquentiel, elle se calcule en nombre de cycles d'horloge. La latence d'un opérateur conditionne la longueur d'un pas de calcul. Elle est également très importante dans les opérations de transmission de données, dans le codage par exemple, où il faut pouvoir exploiter l'information dès qu'elle est disponible. Souvent, on l'exprime en seconde et non plus en cycle d'horloge selon:

$$latence_{temps} = T * latence_{cycle}. \tag{2.2}$$

2.5.4 Puissance dissipée

L'émergence des ordinateurs personnels et des systèmes de communication sans fil exige une faible consommation pour réduire les dispositifs de refroidissement et d'alimentation. Les sources de **dissipation** sont le courant de fuite, les courants parasites, le courant de court-circuit et le courant de charge et de décharge des capacités. Les deux premiers termes sont statiques et négligeables. Les deux derniers sont dynamiques, c'est-à-dire actifs lors d'une transition logique uniquement. Si les tailles des portes sont sélectionnées de telle manière que les temps de transition en entrée et en sortie sont égaux, le courant de court-circuit ne représente que 15% de la consommation dynamique, mais peut être plus important autrement. La source dominante est le chargement et le déchargement des capacités en sortie:

$$P = \frac{1}{2} C_{totale} V_{dd}^2 f_{clk} E_{sw}, \tag{2.3}$$

où V_{dd} est la tension d'alimentation, f_{clk} est la fréquence d'horloge, C_{totale} est la somme de toutes les capacités C_{int}, C_{cell}, C_{inter} (capacités intrinsèques, des cellules attaquées et des interconnexions respectivement), et E_{sw} est le nombre moyen de transitions par période $\frac{1}{f_{clk}}$. En remplaçant le courant de court-circuit par une capacité équivalente virtuelle [9], l'ensemble de la consommation dynamique peut être

estimée par (2.3). Outre la tension d'alimentation, qui peut être abaissée avec un accroissement du chemin critique, le terme à minimiser est la capacité de commutation $C_{sw} = C_{totale}E_{sw}$.

L'activité de commutation d'une porte du circuit combinatoire est difficile à estimer par le nombre de paramètres structurels ou technologiques qui ne sont pas toujours facilement exprimés. On distingue principalement:

- **le modèle de délai**. Le choix d'un modèle de délai permet d'estimer les transitions logiques, c'est-à-dire nécessaires à la bonne fonctionnalité du circuit, et les transitions inutiles, celles n'apportant aucune contribution fonctionnelle (*glitch*). Celles-ci représentent environ 20% de l'activité totale [10]. Le **modèle délai zéro** suppose que les transitions à l'entrée du circuit sont propagées instantanément et néglige donc la puissance inutile P_G. Par contre, le **modèle réel** utilise le temps de propagation (2.1) et prend donc en compte toute la puissance de commutation $P_L + P_G$. La figure (2.7) montre l'influence du modèle de délai sur l'activité de commutation: en délai zéro, les transitions en entrée sont instantanément propagées en sortie de la porte *Or* et aucune transition n'est observable puisque le niveau logique reste à 1. Par contre, lorsqu'on tient compte du délai de l'inverseur, la transition sur la connexion inférieure est plus longue à se propager que celle sur la connexion supérieure. Pendant cette intervalle de temps, la sortie peut effectuer une transition logique complète.

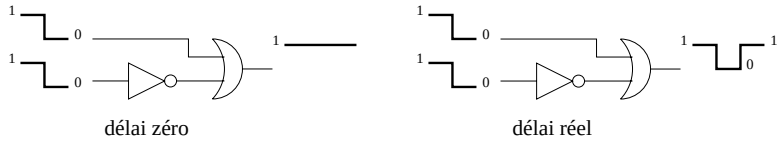

Figure 2.7: Influence du modèle de délai sur l'activité de commutation.

- **la fonctionnalité des cellules**. Elle conditionne la propagation des commutations des entrées à la sortie de la cellule. Certaines portes sont plus sensibles que d'autres; par exemple, si on applique les 16 motifs possibles de transition à l'entrée d'une porte *Nand* et d'un *Xor* à deux entrées, la première commute 6 fois tandis que la seconde 8 fois.

- **la dépendance spatio-temporelle**. Une dépendance spatiale est une contrainte entre les entrées du circuit pour une période d'horloge donnée, et une dépendance temporelle est une contrainte sur une entrée pour une succession de périodes. Lorsque le circuit combinatoire est utilisé en sortie d'une machine d'états finis, les entrées peuvent être fortement corrélées spatialement et temporellement.

- **la structure du circuit**. Par les sortances reconvergentes notamment, elle entraîne une dépendance interne du circuit qui peut modifier fortement l'activité de commutation.

2.6 Techniques d'estimation de la puissance dissipée

Le lecteur peut lire Pedram [11] pour avoir une classification exhaustive des techniques d'estimation et de minimisation de la puissance dissipée. Celles-ci se décomposent en deux classes importantes: les techniques à base de simulation et celles probabilistes.

2.6.1 Techniques à base de simulation

Les techniques à base de simulation, introduites par Tyagi [12], comptabilisent les commutations pour un ensemble de vecteurs d'entrée. Elles sont précises, supportent plusieurs modèles de dissipation et de délais de propagation, mais sont très vite limitées par le temps de simulation et la mémoire requise, dû au nombre exponentiel de vecteurs en fonction du nombre d'entrées. Burch [13] réduit le nombre de stimuli en choisissant de façon aléatoire les stimuli d'entrée. Les mesures consécutives, considérées comme une variable aléatoire, convergent en une densité normale selon le théorème central limite. Le nombre de stimuli à appliquer dépend de l'intervalle de confiance que l'on se donne. Malheureusement, ces techniques précises (voire exactes) ne s'appliquent que sur un circuit combinatoire déjà décomposé, donc ne sont pas applicables pour la sélection des cellules dans la phase de recouvrement.

2.6.2 Techniques probabilistes

L'estimation de l'activité de commutation E_{sw} au nœud n suppose l'estimation de la probabilité $P(n)$ que le signal au nœud n soit au 1 logique. En supposant une indépendance temporelle, nous avons:

$$E_{sw} = 2.P(n)[1 - P(n)], \tag{2.4}$$

correspondant, pour un dèlai zéro, à la somme des probabilités de transition de 1 à 0 et de 0 à 1. Le graphe de décision binaire [14] (BDD) représentant la fonction au nœud n permet de calculer $P(n)$ en un temps linéaire par rapport au nombre de nœuds du BDD [15]. La fonction $y = ab + c$ se représente par le graphe donné en figure (2.8) dans laquelle nous avons ordonné les variables selon a, b, puis c. De la première forme de Shannon: $y = x_i f_{x_i} + \bar{x}_i f_{\bar{x}_i}$, on effectue une traversée du sommet aux feuilles du BDD en calculant:

$$P(y) = P(x_i)P(f_{x_i}) + P(\bar{x}_i)P(f_{\bar{x}_i}). \tag{2.5}$$

Pour le BDD de la figure 2.8, on peut par exemple calculer la probabilité que le nœud racine soit à 1: $P(y) = P(a)P(f_a) + P(\bar{a})P(c)$, $P(f_a)$ étant aussi décomposé selon $P(f_a) = P(b) + P(\bar{b})P(c)$. Lorsque le BDD est trop grand, Kapoor propose dans [16] de partitionner le circuit et de remplacer les partitions indépendantes (ou peu corrélées) par une nouvelle entrée. Le but est bien sûr de limiter le nombre d'entrées maximal des BDD.

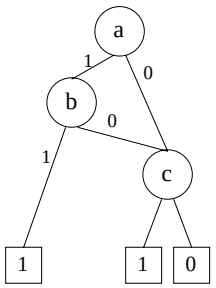

Figure 2.8: BDD de l'expression $y = ab + c$.

Pour prendre en compte les transitions inutiles, Ghosh propose dans [17] la simulation symbolique qui consiste à propager les probabilités de commutation à travers le circuit en utilisant un délai réel. Chaque porte possède une fonction symbolique décrivant l'ensemble des conditions nécessaires à une commutation. La somme de ces probabilités donne une activité moyenne de commutation. Cette méthode comporte les mêmes désavantages que les méthodes à base de simulation.

Les méthodes probabilistes sont moins précises que la simulation mais peuvent être utilisées de manière prévisionnelle pour comparer les activités des différentes cellules et donc être incorporées dans un processus de sélection.

2.7 Influences des critères d'optimisation

Tous les niveaux de la synthèse ou presque peuvent influencer les critères d'optimisation. Toutefois, chacun d'entre eux dépend fortement d'une ou deux opérations spécifiques:

- La latence est fixée au niveau comportemental par l'ordonnancement des opérations. Le choix d'opérateurs multicyle dans la phase d'allocation peut également rallonger la latence. Une fois les pas de calcul décomposés en cycles d'horloge, la latence est figée.

- La surface, tout comme la puissance dissipée et la fréquence d'hologe, dépend de la sélection des cellules de bibliothèques dans l'optimisation logique. Toutefois,

la surface résulte principalement de l'architecture choisie au niveau comportemental et de son degré de parallélisme. Le choix des ressources (allocation comportementale ou macrogénération RTL) influence la surface par le parallélisme interne de chaque opérateur.

- La fréquence d'hologe dépend des opérateurs que l'on place dans chaque cycle: le chaînage d'opérations dans la synthèse comportementale puis le choix des architectures dans l'allocation des ressources (comportementale et RTL) fournissent des équations qui ne pourront être que partiellement optimisées dans la phase de synthèse logique. Notons aussi que le délai des interconnexions intervient à présent pour une partie importante du chemin critique.

- Enfin, la puissance dissipée diminue avec la latence puisque moins d'opérations sont à effectuer. On peut également déconnecter des opérateurs inactifs pour annuler leur consommation. Pour le reste du flot, la puissance dissipée va souvent de pair avec la surface puisque celle-ci est fonction de la capacité totale attaquée.

2.8 Conclusion

Dans ce chapitre, nous avons présenté le flot complet de la synthèse de circuits intégrés suivant les principaux niveaux d'abstraction. Les exigences de productivité obligent le concepteur à choisir un niveau de description plus élevé et à réutiliser le plus grand mombre de blocs possibles. Nous allons donc, dans un premier temps, étudier la conception des blocs réutilisables, puis les méthodologies de conception et de réutilisation.

L'objectif de la synthèse est de satisfaire les contraintes ou les critères d'optimisation du concepteur. Aussi avons nous présenté quatre critères d'optimisation importants que sont la surface, la fréquence d'horloge, la puissance dissipée et la latence.

Chapitre 3

La décomposition technologique

3.1 Introduction

La synthèse logique génère un réseau de portes à partir d'une description logique sous forme d'équations booléennes. Que l'on utilise une synthèse 2-couches ou multi-couches, les équations optimisées sont décomposées sur la bibliothèque de cellules dont le concepteur dispose. Les PAL, de l'anglais *Programmable Logic Array*, permettent d'implanter des expressions booléennes décrites sous la forme de sommes de monômes. Toutefois, alors qu'une synthèse 2-couches est souvent réservée aux implantations de type PAL, une synthèse multi-couches est presque toujours choisie pour la cible de cellules standards. Cette étape de **décomposition technologique** ne modifie pas la structure des équations, mais choisit un ensemble optimal de portes pour minimiser le ou les critères d'optimisation. Cette partie seulement s'apparente à une méthode de réutilisation grâce à une utilisation intensive de la bibliothèque de cellules standards. La figure 3.1 illustre l'étape de réutilisation de la bibliothèque: chaque porte peut être inférée plusieurs fois en fonction de ses performances. Celles-ci sont évaluées par une fonction de gain, qui peut être la surface, le délai ou la puissance dissipée, afin d'être utilisées par l'outil de décomposition technologique. Celui-ci sélectionne alors un ensemble de portes de la bibliothèque afin de recouvrir les équations booléennes optimisées.

Nous limitons donc notre étude de la synthèse logique à la phase de décomposition technologique. Contrairement aux chapitres suivants, dominés par la construction d'une bibliothèque de composants réutilisables, le concepteur n'a pas accès à la bibliothèque au niveau logique. Celle-ci est en effet entièrement fournie par le fondeur. Par conséquent, la contribution est ici d'améliorer l'outil de décomposition technologique afin de mieux utiliser les ressources disponibles. En particulier, nous nous efforcerons:

- de proposer une méthode efficace d'utilisation des portes les plus complexes qui sont également celles de plus fort gain. La complexité varie d'une bib-

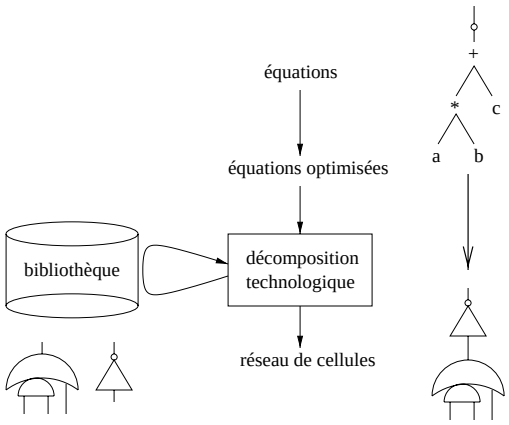

Figure 3.1: L'étape de réutilisation en synthèse logique.

liothèque à l'autre, mais le bénéfice des cellules complexes lié aux améliorations technologiques rendent ces portes très populaires.

- de proposer une méthode de sélection des cellules afin d'optimiser la puissance dissipée, le seul critère qu'il reste à approfondir. En particulier, nous optimiserons la puissance totale dissipée plutôt que la puissance logique.

L'amélioration de l'outil de synthèse en vue d'une utilisation accrue de la bibliothèque est la seule possibilité qui nous est offerte d'envisager une approche de réutilisation pour l'étape de synthèse logique. Notre approche est bien entendu applicable à tous les critères présentés au chapitre 2. Toutefois, elle prend une dimension particulière pour le critère de la puissance dissipée puisque, comme nous le verrons par la suite, il s'agit du critère principal d'utilisation de cellules complexes. L'aspect réutilisation y est donc bien illustré.

Le chapitre est composé comme suit: la section 2 présente l'état de l'art des algorithmes classiques de décomposition technologique, que ce soit les isomorphismes ou la sélection. Puis, nos approches d'isomorphisme et de recouvrement sont présentées: du problème rencontré, nous proposons une solution efficace étayée de résultats expérimentaux.

3.2 Les algorithmes classiques

3.2.1 Le recouvrement

La décomposition technologique peut être réalisée par deux types d'approches: les **algorithmes dynamiques** [18][19] et les **systèmes à bases de règles** [20][21]. Les méthodes à base de règles supposent qu'une première réalisation, même indépendante

de la technologie, existe. Des transformations locales (ou règles) préservant la fonctionnalité du circuit sont appliquées afin d'optimiser les réalisations successives. *Lss* [20] adopte une stratégie gloutonne qui consiste à examiner l'ensemble des règles qui peuvent être appliquées et à sélectionner celle qui amène le plus fort gain, et cela jusqu'à obtention d'un **minimum local**. *Socrates* [21] examine un nombre fixe de réalisations correspondant à l'application de plusieurs règles successivement et sélectionne la meilleure combinaison. Ces méthodes souffrent de résultats imprévisibles dûs à la recherche de minimums locaux et sont très coûteuses en temps d'exécution et en mémoire, à cause notamment de la génération des règles pour les différentes bibliothèques cibles.

Les heuristiques dynamiques utilisent les techniques de génération de code dont le but est de sélectionner l'ensemble d'instructions conduisant au temps d'exécution minimum [22]. Le problème de la couverture d'un graphe orienté acyclique (DAG) est NP-complet et peut être très bien réalisé par la couverture optimale d'une forêt d'arbres correspondant à l'ensemble des graphes obtenus en coupant les nœuds à sortances multiples. Chaque cellule retenue réalise une partie de la fonction à décomposer et on lui associe un coût. *Dagon* [19] et *Mis* [18] divergent quant à la représentation interne des cellules et de la fonction, mais utilisent la programmation dynamique pour sélectionner l'ensemble optimal de cellules. Ces heuristiques recherchent un **minimum global**.

Dans ces deux approches, deux sous-problèmes doivent être résolus: l'**isomorphisme** (*matching*) et la **sélection**. L'isomorphisme signifie reconnaître qu'une portion du circuit combinatoire peut-être réalisée par une cellule donnée. La sélection signifie choisir l'ensemble des cellules qui optimise le critère de décomposition.

3.2.2 Les isomorphismes

Les isomorphismes sont structurels ou booléens. Les **isomorphismes structurels** représentent le réseau booléen (circuit) et les cellules de bibliothèque par des fonctions booléennes simples, par exemple en *Nand* à deux entrées et en inverseurs. Puisque plusieurs décompositions peuvent exister, toutes les représentations sont supposées être énumérées. L'algorithme d'isomorphisme consiste donc à détecter un isomorphisme entre le graphe de l'une des représentations d'une des cellules et un sous-graphe du réseau. La figure 3.2 présente deux décompositions d'une même cellule de bibliothèque (*Nand* à 4 entrées). Elles se distinguent par une structure différente de l'arbre booléen, à base de portes *Nand2* et d'inverseurs, qui les représente: la porte de gauche a une structure équilibrée tandis que celle de droite est totalement déséquilibrée. Une porte plus complexe peut engendrer plusieurs centaines ou milliers de décompositions. Ces approches souffrent du fait que les représentations ne sont pas canoniques et donc que des isomorphismes potentiels peuvent ne pas être détectés.

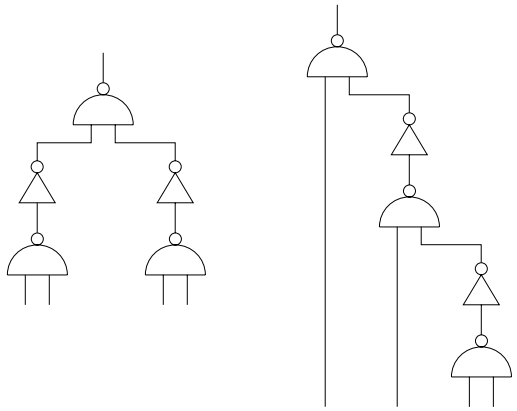

Figure 3.2: Deux repésentations d'un *Nand4*.

Les **isomorphismes booléens** sont basés sur une représentation booléenne (canonique) des fonctions logiques. Considérons une fonction $f(x)$ avec n variables d'entrées de vecteur x. Soit $g(y)$ une fonction de bibliothèque de m variables de vecteur y. On suppose $n = m$ bien que, par collage ou en utilisant des conditions *don't care*, n et m peuvent être différents. f et g sont équivalentes si

$$f(x) \bar{\oplus} g(x) \;=\; 1 \tag{3.1}$$

En acceptant l'altération de la polarité des entrées et de la sortie ainsi que la permutation des n entrées, f et g sont NPN-equivalentes s'il existe une opération de permutation P et des opérations de complémentation N_i, N_o tels que $f(x) = N_o.g(PN_i.x)$ soit une tautologie. En principe, $2^{n+1}n!$ verifications doivent être réalisées, chacune d'entre elles pouvant être effectuée en temps constant par équivalence de BDD [23]. Pour réduire ce nombre de vérifications, des méthodes canoniques, à base de signatures, ou spectrales sont appliquées:

- **les méthodes canoniques** introduites par Burch dans [24] consistent à mettre une fonction sous une forme canonique C_{NPN} représentant sans équivoque une classe de NPN-équivalence. La comparaison de $C_{NPN}(f)$ et de $C_{NPN}(g)$ suffit pour détecter un isomorphisme.

- **Les signatures** sont des représentations compactes qui caractérisent quelques propriétés de la fonction. Puisqu'une signature peut être liée à plusieurs fonctions, ces méthodes sont potentiellement moins performantes que les formes canoniques. Les symétries, la taille des co-facteurs peuvent néanmoins permettrent de filtrer les fonctions pour lesquelles il ne peut y avoir d'isomorphisme. Nous développons ci-dessous brièvement une méthode de signature (Actel [25]) que nous avons incorporée dans un outil de synthèse.

- Enfin, **les méthodes spectrales** caractérisent complètement chaque fonction. Puisque les opérations appliquées aux fonctions booléennes ont des effets spécifiques sur une ou plusieurs entrées du spectre, celui-ci suffit pour détecter l'équivalence via les opérateurs N_i, P, P_o.

Voici une méthode de signature efficace (qui, en fait, a les propriétés d'une forme canonique). Le lecteur est encouragé à lire [26] pour plus d'information. Soit une table de vérité dont une forme dite canonique est obtenue lorsque les lignes sont rangées selon les valeurs binaires des variables d'entrées, et considérons la sortie comme un nombre binaire appelé nombre-identité. Deux fonctions sont P-équivalentes si elles ont le même nombre-identité minimum. L'idée est de distinguer les entrées de la table de vérité de manière à ne pas tester les $n!$ permutations possibles. Un invariant, associé à un objet de la table (ligne, colonne ou variable) est défini comme une valeur inchangée par les échanges de lignes ou de colonnes (par exemple le poids d'une ligne ou d'une colonne). Nous rangeons donc les variables par ordre croissant d'invariant. Pour les variables de même invariant, on regroupe les variables symétriques et on décide de permuter uniquement les groupes symétriques de même invariant et de même cardinalité puisque ce sont les seuls à ne pas pouvoir être distinguables. Le nombre identité minimum obtenu par l'application de ces permutations est également caractéristique d'une classe de P-équivalence. Le bénéfice de tels regroupements est que le nombre de tables de vérité à générer est presque constant comme le montre le tableau 3.1 réalisé sur une bibliothèque de FPGA de la société ACTEL: pour la grande majorité des cellules, une seule table de vérité (TdV) doit être construite, alors que près de 500 d'entre elles étaient précédemment requises. Avec peu d'effort, nous réalisons une NP-équivalence [27]. Cette méthode servira de comparaison avec les techniques d'isomorphismes développées plus loin.

Bibliothèque	nb portes	TdV sans invariant	TdV avec invariant
Actel1	104	52471	127
Actel2	130	48931	150
Actel3	131	53993	154

Tableau 3.1: Nombre de tables de vérité.

3.2.3 La sélection

Le recouvrement consiste à **sélectionner** un ensemble d'isomorphismes réalisant totalement le réseau booléen. Alors que les isomorphismes permettent d'accroître l'espace des solutions, le processus de sélection a pour but de parcourir cet espace pour sélectionner la meilleure (ou une bonne) solution. Pour cela, les méthodes **gloutonnes** choisissent, en parcourant le circuit combinatoire non recouvert, l'isomorphisme qui amène le plus fort gain. Elles n'atteignent malheureusement que des minimums

locaux puisque la meilleure solution peut passer par le choix d'un isomorphisme de faible gain mais dont la sélection conduit, par la suite, à réaliser un ensemble d'isomorphismes de gain plus important.

Par contre, la solution optimale est déterminée par le parcours exhaustif de l'espace des solutions, c'est-à-dire tester toutes les combinaisons possibles d'isomorphismes réalisant la fonction logique. La **programmation dynamique** nous permet de parcourir toutes les solutions de manière intelligente en ne déterminant la meilleure combinaison en chaque nœud du circuit qu'une seule fois. Soit le graphe de la fonction à décomposer représenté sur la figure (3.3) qui utilise les mêmes notations que les formules suivantes. On y a représenté un isomorphisme, c'est-à-dire une équivalence logique entre une portion d'un arbre booléen et une cellule de bibliothèque.

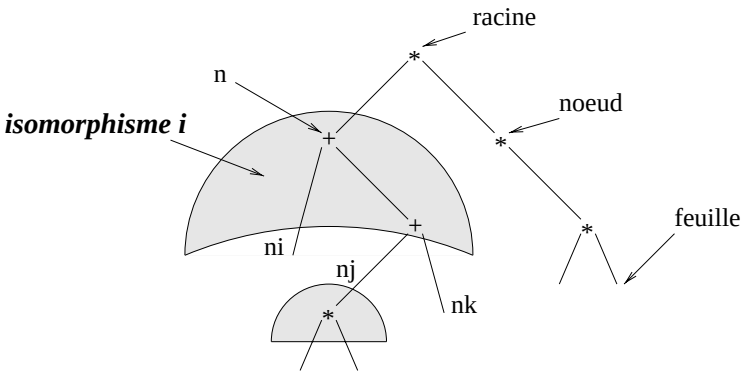

Figure 3.3: Graphe du réseau à décomposer.

Lorsque les coûts des isomorphismes sont indépendants, on peut déterminer en un temps constant le coût d'un isomorphisme i au nœud n, c'est-à-dire le coût du meilleur recouvrement de sommet n en utilisant l'isomorphisme i. En notant n_i les entrées de l'isomorphisme i, et $iso(n_i)$ les isomorphismes au nœud n_i:

$$coût(n,i) = coût(i) + \sum_{n_i \in entrée(i)} min_{j \in iso(n_i)}(coût(n_i, j)). \tag{3.2}$$

La solution du recouvrement au nœud n est donc la combinaison pour laquelle le coût est minimal:

$$solution(n) = min_{i \in iso(n)}[coût(n,i)]. \tag{3.3}$$

(3.3) peut donc être reformulée par

$$solution(n) = min_{i \in iso(n)}[coût(i) + \sum_{n_i \in entrée(i)} solution(n_i)]. \tag{3.4}$$

A chaque nœud visité lors d'un parcours des feuilles à la racine, (3.4) permet de calculer le meilleur recouvrement en ce nœud en calculant le coût de chaque isomorphisme. Le meilleur recouvrement à la racine du graphe est la solution du problème.

Lorsque les coûts des isomorphismes sont dépendants, comme l'est le délai dont le coût dépend de la charge et donc des isomorphismes des nœuds sortants, (3.3) ne permet plus de déterminer la meilleure solution en chaque nœud. En effet, la meilleure solution dépend non seulement de l'isomorphisme mais aussi de la charge en amont. Deux passes sont requises: dans la première passe à travers le graphe des feuilles à la racine, chaque nœud est visité une seule fois et le coût minimum en chaque nœud est mémorisé pour chaque valeur de la charge. Dans la seconde passe, de la racine aux feuilles, le meilleur isomorphisme est sélectionné pour chaque nœud basé sur les enregistrements précédents. On peut également mémoriser le meilleur isomorphisme pour un intervalle de valeurs plutôt que pour toutes les charges possibles (Touati [28]). Plus la dépendance est profonde (2 niveaux ici), plus le processus de programmation dynamique est coûteux.

Pour la capacité de commutation, le coût dépend aussi de la charge par le terme $C_{cell}E_{sw}$, où C_{cell} est la capacité des cellules attaquées. Mais celui-ci est le même pour tous les isomorphismes puisque E_{sw} ne dépend pas des nœuds sortants (et pas de l'isomorphisme). Ainsi, le meilleur isomorphisme peut être déterminé sans que les isomorphismes des nœuds sortants ne soient connus.

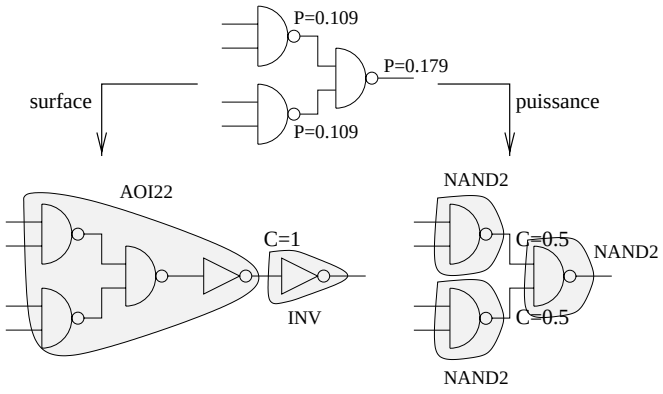

Figure 3.4: Recouvrements pour les options surface et puissance.

Considérons le circuit de la figure 3.4 comme un exemple où la différence entre recouvrement pour la surface et la puissance dissipée est significatif. Les probabilités de commutation aux sorties des portes G_1, G_2 et G_3 sont 0.109, 0.109 et 0.179 respectivement. On utilise une bibliothèque simple de portes *Nand2*, *Inv* et *AOI22* de surfaces respectives 1, 0.5 et 2, et de capacités d'entrée respectives 0.5, 1 et 1. Un

nœud est dit actif lorsque la capacité de commutation, c'est-à-dire le produit de la probalilité de commutation et de la capacité attaquée, est importante par rapport aux autres nœuds. Pour le recouvrement surface, on utilise le moins de portes possibles quitte à laisser apparents deux nœuds très actifs, de capacité de commutation 0.179. Par contre, le recouvrement puissance vise à masquer les nœuds très actifs et à ne laisser apparents que les nœuds peu actifs, de capacité de commutation $\frac{0.109}{2}$. Ainsi, la recherche du minimum peut considérablement modifier la stucture du circuit en fonction du critère d'optimisation.

3.3 Isomorphisme de portes complexes

3.3.1 Position du problème

Les critères de surface et de puissance dissipée sont sensibles à la taille des portes de bibliothèque utilisées parcequ'une **cellule complexe** contient jusqu'à deux fois moins de transistors qu'un ensemble de portes équivalentes plus petites, sans utiliser de capacités internes. La surface d'une telle cellule, pondérée par le nombre de nœuds élémentaires contenus, est donc faible. De plus, une cellule complexe masque plus de nœuds que plusieurs cellules équivalentes donc dissipe très peu de puissance. La figure 3.5 montre comment une porte complexe, en remplaçant trois plus petites, est à la fois moins grosse et dissipe beaucoup moins de puissance puisqu'elle ne laisse apparent qu'un seul nœud: à gauche, les trois petites portes laissent apparents trois nœuds de probabilité de commutation totale $3.P_t$. Par contre, à droite, un seul nœud est apparent tandis que les autres sont masqués par la cellule. La puissance dissipée est donc moindre.

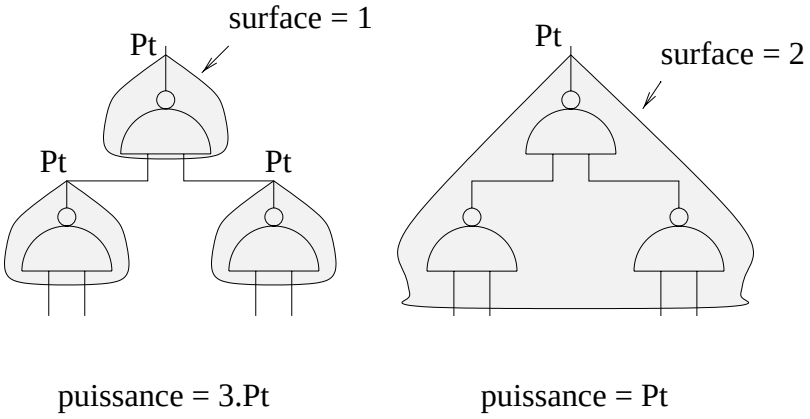

Figure 3.5: Utilité des portes complexes.

Il est reconnu que le recouvrement vitesse n'utilise pas les portes complexes car le chemin critique réussit toujours à passer par le chemin de profondeur maximale.

Il faut se rappeler, en effet, que le chemin critique n'est pas une notion additive mais se définit comme le chemin de délai maximal. A moins de couvrir toutes les équations booléennes par un ensemble de portes complexes, et même si plusieurs chemins empruntent des portes complexes, il existe au moins un chemin n'utilisant que des petites portes. Par conséquent, ce chemin devient le chemin critique. Nos résultats expérimentaux ont par ailleurs confirmé ceci.

L'impact d'une porte complexe est donc visible au niveau de l'implantation physique du circuit, par le nombre réduit de transistors utilisés et le masquage de nœuds actifs. Les méthodes classiques de synthèse de *layout* utilisent les approches *full-custom* ou à base de cellules standards. La première, qui consiste à dessiner les cellules à la main, pour un résultat très dense, n'est guère plus utilisée pour des raisons évidentes de temps de conception. La seconde utilise une bibliothèque de cellules *full-custom* déjà caractérisées à placer-router de manière automatique. Il existe une troisième méthode innovante pour la synthèse automatique de macro-cellules. Elle consiste à générer le dessin des masques des cellules, selon un style régulier, réalisant ainsi une assignation technologique d'une bibliothèque virtuelle de portes complexes. Les logiciels TABA et TROPIC permettent donc [88][89][90]: 1) de générer toute sorte de portes complexes et pas seulement celles disponibles dans une bibliothèque de cellules standards, 2) de réaliser une synthèse physique indépendante de la technologie, caractéristique nécessaire pour suivre les avancées technologiques rapides. Ceci permet d'effectuer une migration technologique facile en changeant l'ensemble des règles de dessin. Les logiciels permettent aussi 3) d'ajuster la taille des transistors en fonction des contraintes temporelles de telle sorte que le délai le long du chemin critique soit diminué, et 4) de s'affranchir du coût important de la maintenance d'une librairie de cellules et de caractérisation. Cette approche se distingue de celle que nous présentons ici par le niveau d'abstraction où elle se situe: la génération automatique de masques à partir d'une bibliothèque virtuelle nécessite la phase d'assignation technologique, alors que nous nous limitons à la représentation booléenne des cellules de bibliothèques. Nous sommes donc tributaires de la bibliothèque choisie et de ses caractéristiques pré-définies, et nous nous efforçons d'améliorer la phase de synthèse logique uniquement. Les possibilités de gain, comme indiquées en fin de chapitre, sont plus faibles mais sont incorporables dans un outil classique de synthèse.

Les isomorphismes stucturels supposent la construction de toutes les représentations possibles des cellules de bibliothèques en utilisant par exemple les portes élémentaires *Nand2* et *Inv*. Cette approche n'est plus possible pour les portes complexes puisque le nombre de décompositions croît exponentiellement en fonction du nombre d'entrées; elle pose problème tant en temps d'exécution qu'en ressource mémoire. La nécessité de construire la bibliothèque de cellules décomposées rend les isomorphismes structurels inefficaces pour les portes complexes.

Les isomorphismes booléens n'utilisent qu'une seule représentation des cellules. Toutefois, résoudre (3.1) nécessite de disposer de f, sous-fonction de l'expression à décomposer. Trouver tous les isomorphismes possibles pour une cellule de bibliothèque de m entrées au nœud n nécessite d'extraire du graphe toutes les fonctions de m entrées correspondant à tous les sous-graphes de m feuilles au nœud n. Cette procédure d'extraction est résolue en sélectionnant m points de coupes. La figure 3.6 montre plusieurs insertions de points de coupe dans un réseau booléen afin d'extraire des sous-graphes de 4 entrées: l'arbre étant équilibré, l'outil de décomposition a le choix entre parcourir la sous-arbre de gauche ou le sous-arbre de droite jusqu'à concurrence de quatre feuilles. On trouve donc ici cinq ensembles de points de coupe.

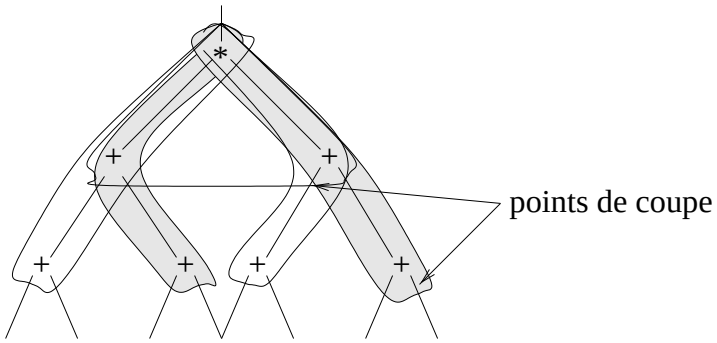

Figure 3.6: Extraction de sous-graphes par insertion de points de coupe.

Or le nombre d'extractions croît exponentiellement en fonction du nombre de points de coupe, plus fortement que pour les isomorphismes sructurels, notamment dû à l'insertion de paires d'inverseurs, dont le but est d'accroître l'espace des solutions en permettant les isomorphismes de cellules à entrées ou sorties complémentées. Cette procédure d'insertions multiplie par 2^m le nombre d'extractions à effectuer. Un algorithme d'insertions de points de coupe est fourni dans [26] pour les critères surface (ou puissance) et vitesse. L'option surface effectue une extraction exhaustive des sous-graphes de m entrées au plus, tandis que l'option vitesse effectue un parcours en profondeur de l'arbre puisque le chemin critique passe par le réseau de cellules le plus profond. Il est donc préférable de limiter la profondeur du recouvrement.

3.3.2 Solution proposée

La décomposition des cellules ou l'extraction des sous-graphes est une opération structurelle préliminaire à la procédure d'isomorphisme. Le problème est qu'il n'y a pas de contrôle possible entre cette phase structurelle et la phase d'isomorphisme, donc toutes les décompositions ou toutes les extractions doivent être obtenues pour que la procédure d'isomorphisme accepte ou rejette le candidat. L'idée est de réunir la phase d'isomorphisme et la phase stucturelle dans une même opération. Le but est

de pouvoir contrôler la phase structurelle en fonction des indications fournies par la procédure d'isomorphisme. La procédure d'isomorphisme ne peut-être donc que structurelle. Elle est moins performante que son équivalente booléenne mais son utilisation dans un flot complet de décomposition est beaucoup plus efficace. L'approche ci-dessous ne cherche que les décompositions qui conduisent à un isomorphisme. Du nombre maximal de décompositions, on ne génère maintenant que le nombre minimal d'entre elles.

Une fonction booléenne est une expression booléenne parenthésée représentée par un graphe orienté acyclique. Les nœuds du graphe sont les opérateurs *Et*, *Ou* ou *Inv* et les feuilles sont les littéraux représentants soit des entrées primaires, soit des sous-fonctions. Deux graphes sont équivalents si leurs fonctions booléennes associées sont identiques. Un sous-graphe est la restriction du graphe à un sous-ensemble de ses nœuds.

Definition 3.3.1 (Expansion de nœud) *Soit un nœud n* Et *(respectivement* Ou*) d'un graphe. Une expansion de n est une décomposition de n en nœuds* Et *(respectivement* Ou*) binaires.*

Definition 3.3.2 (Réduction de nœud) *Soit un ensemble de nœuds* Et *(respectivement* Ou*) consécutifs (i.e., reliés par une relation père-fils). La réduction de cet ensemble consiste à le remplacer par un nœud n-aire* Et *(respectivement* Ou*) équivalent.*

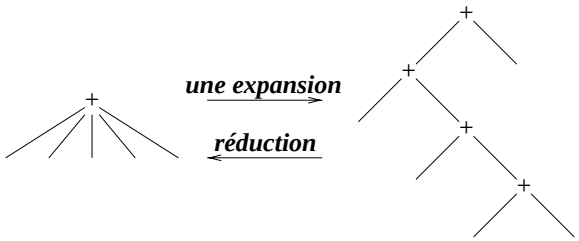

Figure 3.7: Expansions et réduction.

La réduction est l'opération inverse de l'expansion, présentées en figure 3.7: l'expansion de droite n'utilise que des nœuds binaires tandis que la réduction de gauche rassemble tous les nœuds du même type pour former un nœud de taille maximale. La présence d'inverseurs entre les nœuds *Et* et *Ou* conduit à de nombreuses représentations grâce aux lois de DeMorgan. Deux d'entre elles sont importantes pour lesquelles les nœuds internes sont dépourvus d'opérateurs *Inv*.

Definition 3.3.3 (Représentation naturelle d'un graphe) *La représentation naturelle consiste à repousser les inverseurs aux feuilles du graphe, grâce aux lois de DeMorgan.*

Definition 3.3.4 (Représentation inversée d'un graphe)
la représentation inversée consiste à introduire dans la représentation naturelle du graphe une paire d'inverseurs et à en propager un jusqu'aux feuilles.

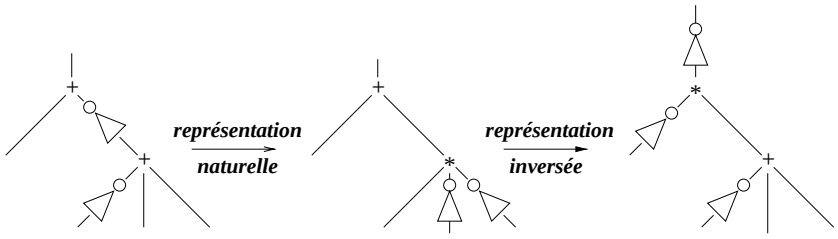

Figure 3.8: Représentations naturelle et inversée.

La figure 3.8 montre ces deux représentations qui consistent à enlever les inverseurs des nœuds internes: au centre, les inverseurs sont repoussés aux feuilles de l'arbre, tandis qu'à droite, un des inverseurs est laissé à la racine de l'arbre et les autres aux feuilles. Le graphe résultant est soit complémenté (représentation inversée), soit direct (représentation naturelle). Le recouvrement consiste à choisir une représentation commune pour le graphe de la fonction et des cellules de bibliothèque afin de détecter les isomorphismes. On choisit pour le graphe l'expansion de la représentation naturelle qui minimise la profondeur des nœuds binaires. Ce choix a pour but de diminuer le chemin critique. On choisit pour les cellules la forme réduite des représentations naturelles et inversées. En effet, l'utilisation des opérateurs *Nand2/Inv* permet de représenter les nœuds *Et* et *Ou*. A présent, la représentation distincte des nœuds *Et* et *Ou* nous oblige à gérer les deux formes équivalentes de DeMorgan. Le contrôle que nous effectuons sur le choix des décompositions vient de la décoration que nous apportons aux cellules et au graphe de la fonction à décomposer:

Definition 3.3.5 (Décoration des cellules) *La décoration d'un nœud* Et *ou* Ou *est le nombre de nœuds binaires contenus dans une expansion de ce nœud. Il s'agit du nombre de ses fils moins un.*

Definition 3.3.6 (Décoration du graphe) *La décoration d'un nœud binaire est le nombre maximal de nœuds binaires consécutifs de même type contenus dans le sous-graphe de racine ce nœud. Elle se calcule par la récursion* $déco(n) = 1 + \sum_{v_i} déco(v_i)$, *où* v_i *sont les fils de même type que* n.

Nous cherchons à présent à recouvrir un sous-graphe du réseau booléen par un nœud d'une cellule de bibliothèque (pas encore par une cellule entière). Pour détecter un isomorphisme potentiel, nous acceptons provisoirement les cellules pour lesquelles le nœud est décomposable dans le graphe et nous éliminons les autres. La condition pour qu'un tel recouvrement soit possible, mais pas encore certain, est la suivante:

Propriété 3.3.1 (Condition d'isomorphisme) *Soit un nœud g du graphe à recouvrir par un nœud c d'une cellule de bibliothèque. Un isomorphisme est possible entre ces deux nœuds si et seulement si:*

$$\begin{cases} type(g) & = type(c) \\ déco(c) & \leq déco(g). \end{cases} \tag{3.5}$$

En effet, si la décoration de la cellule est inférieure à celle du graphe, alors il existe une expansion de la cellule isomorphe au sous-graphe de racine g. Dans le cas contraire, au moins un des nœuds de la cellule ne trouvera pas un nœud du graphe du même type. La topologie du graphe dirige la décomposition des cellules: les seules décompositions générées sont celles qui conduisent à un isomorphisme.

Une cellule est constituée de plusieurs niveaux de nœuds. (Par exemple, un *AO21* est constitué d'un nœud *Ou* suivi d'un nœud *Et*.) L'algorithme cherche toutes les expansions du nœud racine. Pour celles-ci, il associe les nœuds fils de la cellule avec les sous-graphes de l'expansion. Pour chaque association, les deux phases successives (expansion et association) sont répétées jusqu'aux feuilles de la cellule. Toutes les combinaisons d'expansions et d'associations sont les solutions de l'algorithme. Une cellule réalisant un isomorphisme possède forcément une telle décomposition et une de ces décompositions conduit à un isomorphisme puisque les deux graphes sont isomorphes.

3.3.3 Expansions et Associations

Générer une expansion consiste à former un graphe de nœuds binaires. Ce problème est résolu par une approche *diviser-pour-mieux-régner* qui consiste à former les sous-graphes droits et gauches du nœud racine, toutes les tailles de ces sous-graphes devant être examinées: pour une décoration de N, on forme le nœud binaire et on cherche les expansions gauches et droites de décoration respectives i et $N-1-i$ pour i variant de 0 à $N-1$. Lorsque la taille d'un sous-graphe est nulle (feuille de la cellule), une feuille de l'expansion est atteinte. Le nombre d'expansions dépend de la structure du graphe. Dans le pire cas, pour un nœud n-aire, on en compte (avec $E_0 = 1$):

$$E = \sum_{i=0}^{N-1} E_i E_{N-1-i} \tag{3.6}$$

Considérons à titre d'exemple la figure 3.9 préalablement décoré, ainsi que la cellule à décomposer. La procédure d'expansion trouve deux expansions du nœud racine (*Ou3*). En effet, trouver une expansion d'un nœud de décoration 2 nécessite de déterminer deux nœuds binaires consécutifs dans l'arbre booléen. On en trouve évidemment deux, suivant que l'outil de décomposition emprunte le sous-arbre gauche ou le sous-arbre droit.

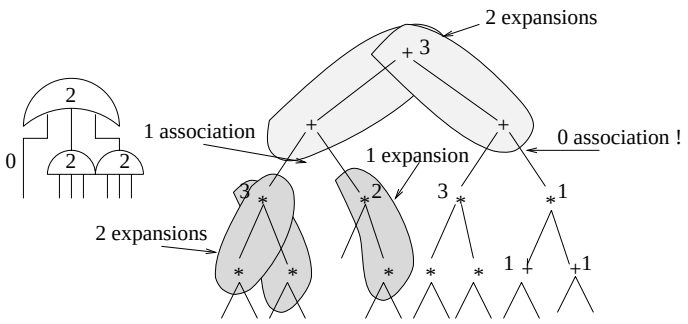

Figure 3.9: Exemple d'isomorphisme.

Nous avons à présent un ensemble de fils de la cellule et un ensemble de sous-graphes à associer pour réaliser un isomorphisme. Les fils de la cellule sont des nœuds n-aires de type différent de celui qui vient d'être décomposé ou sont des feuilles. Les sous-graphes sont des feuilles (décoration 0), ou des nœuds binaires. Lorsque ces nœuds binaires sont du même type que leur père, on leur donne provisoirement la décoration 0, étant traités comme des feuilles. La condition pour laquelle un fils de la cellule peut-être associé à un sous-graphe est que cette association vérifie la condition d'isomorphisme (3.5b) puisque les nœuds sont maintenant de même type.

Definition 3.3.7 (Association) *Soit un ensemble C de fils de cellule et un ensemble G de sous-graphes de même cardinalité. Une association lie chaque élément C_i de C avec un élément différent G_i de G tel que C_i et G_i vérifie la condition d'isomorphisme (3.5).*

Etant donné deux ensembles à associer, la propriété suivante permet de tester l'existence d'une telle association.

Propriété 3.3.2 (Existence d'une association) *Soit les ensembles C et G rangés par décoration croissante. Il existe une association si et seulement si:*

$$\forall i \in [1, N], déco(C_i) \leq déco(G_i). \tag{3.7}$$

Enfin, il nous faut savoir si la construction d'une association peut effectivement se terminer avec succès.

Propriété 3.3.3 (Construction d'une association) *Soit un ensemble C de fils de cellule et un ensemble G de sous-graphes de même cardinalité. On suppose l'existence d'une association A entre les éléments de C de plus forts poids et un sous-ensemble de G quelconque. S'il existe une association entre C et G, alors A peut toujours être complétée en une association entre C et G.*

Cette propriété nous permet de construire itérativement toutes les associations possibles: tout d'abord, l'existence d'une association est vérifiée (3.7). Ensuite, on associe à chaque élément C_i de C une liste L_i de sous-graphes G_j de G tel que la condition d'isomorphisme est vérifiée pour tous les couples (C_i, G_j). On parcourt C en partant de l'élément de décoration maximale. Lorsqu'on traite C_j, on a déjà formé toutes les associations A_{j+1} entre les C_i et G pour $j+1 \leq i \leq N$. Pour chaque élément l de L_j, on forme les associations A_j entre les C_i et G pour $j \leq i \leq N$ en ajoutant aux A_{j+1} le couple (C_j, l). l est éliminé de chaque liste L_i. Lorsque $j = 1$, on a trouvé toutes les associations. Le nombre de ces associations dépend des décorations de la cellule et du graphe. Dans le pire des cas, on en trouve $N!$.

Reprenons notre exemple 3.9. Pour l'expansion de droite, nous avons l'ensemble des sous-graphes de cardinalités $(0, 3, 1)$, la cardinalité nulle provenant du nœud binaire de gauche considéré à présent comme une feuille. Mais les fils de la cellule à décomposer sont de cardinalités $(2, 2, 0)$. La propriété 3.3.2 nous indique qu'il n'existe pas d'association puisque le fils de décoration 2 ne peut s'associer avec le sous-arbre de décoration 1. Par contre, pour l'expansion de gauche, nous avons l'emsemble des sous graphes de cardinalités $(0, 3, 2)$. Pour ce cas, il existe deux associations suivant que l'on permute les deux fils identiques de la cellule. Une fois une phase d'expansions et d'associations effectuée, il ne reste plus qu'à la recommencer jusqu'à obtention des feuilles de la cellule. Pour l'une de ces associations uniquement, deux expansions existent, donc deux isomorphismes seulement sont détectés. Notons qu'une cellule de 7 entrées possède déjà beaucoup plus que deux décompositions.

D'après la construction précédente, l'algorithme permute l'ensemble des feuilles. C'est souvent irréalisable et de plus totalement inutile dans un recouvrement surface ou puissance. Et puisque cette procédure d'isomorphisme n'est pas conçue pour le recouvrement vitesse, qui n'utilise que très peu les portes complexes, nous renonçons à cette permutation. Les cellules contenant un cycle, telles que les multiplexeurs et les *Xors* peuvent être représentées par des graphes sans cycle mais utilisant des feuilles identiques (figure (3.10)). L'algorithme permute ces feuilles. Ensuite, pour chaque isomorphisme réalisé, il faut vérifier que les sous-graphes associés aux feuilles identiques sont isomorphes. Cette méthode permet de palier à l'inconvénient majeur des approches structurelles.

La complexité de notre algorithme est la même que l'isomorphisme structurel dans le pire cas, c'est-à-dire lorsque toutes les décompositions d'une cellule conduisent à un isomorphisme. Notre algorithme doit les trouver et effectuera donc toutes les décompositions nécessaires. Toutefois, la complexité est faible dans le cas moyen, c'est-à-dire lorsque peu de décompositions conduisent à un isomorphisme. Dans ce cas, notre algorithme ne va trouver que celles-ci alors qu'une approche d'isomorphisme structurel essaiera toutes les décompositions possibles.

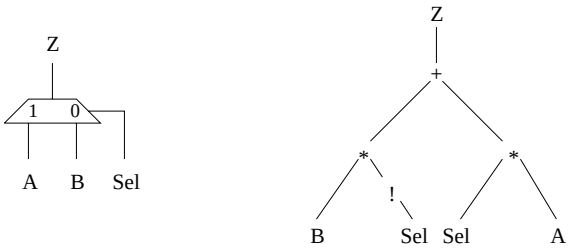

Figure 3.10: Représentation d'un multiplexeur.

3.3.4 Résultats expérimentaux

La procédure d'isomorphisme de portes complexes [29] est comparée aux deux autres procédures d'isomorphismes structurels et booléens précédemment présentées. Les bibliothèques utilisées sont *cg24* and *hg62g* des sociétés japonaises Fujitsu et Hitachi. Elles contiennent quelques portes complexes, certaines d'entre elles possèdent jusqu'à 16 entrées. Les méthodes d'isomorphismes classiques ne peuvent être appliquées puisque le nombre de décompositions et d'extractions est prohibitif. Aussi, nous comparons notre procédure aux heuristiques développées dans l'outil de synthèse ASYL+ [30]:

- l'isomorphisme structurel (Keutzer [19]) est appliqué à une partie seulement des décompositions. Les graphes de profondeur minimale sont générées ce qui permet de représenter presque toutes les décompositions des petites cellules et de ne représenter qu'un nombre constant des décompositions des cellules plus complexes (heuristique 1). Une deuxième heuristique est appliquée qui revient à éliminer toutes les cellules dont le nombre exhaustif de décompositions est trop important, c'est-à-dire les cellules dont le nombre d'entrées est grand (heuristique 2).

- l'isomorphisme booléen (Actel [25]) est appliqué sur une partie seulement des extractions. On utilise la procédure d'extraction exhaustive pour les nombres de points de coupe faibles (au plus 6 points de coupe) et on sélectionne une partie des extractions pour les nombres de points de coupe plus importants. Bien sûr, les isomorphismes des cellules complexes n'ont que très peu de chance d'être détectés puisque le nombre d'extractions retenues est très faible. Notons aussi que la présence de paires d'inverseurs augmente considérablement le nombre d'extractions, qui est plus important que pour la procédure d'isomorphisme structurel.

Comme présenté dans le tableau 3.2, notre procédure optimise le critère puissance (et également le critère surface) de 4% à 6.2% par rapport aux meilleures heuristiques des isomorphismes structurels et booléens. L'estimateur de puissance dissipée est la

Benchmarks	Notre procédure	Isomorphismes structurels 1	Isomorphismes structurels 2	Isomorphismes booléens
5xp1	2247.6	2486.5	2311.2	2352.8
9sym	3081.1	4420.6	3409.4	3563.1
alu2	5595.5	7367.1	5928.6	6074.6
alu4	25389.1	32356.0	27446.9	28673.8
b9	4632.2	6220.2	4925.2	5051.3
bbara	1048.1	1179.3	1091.7	1113.5
bbsse	2115.2	2495.4	2203.3	2237.1
bbtas	423.0	430.9	425.6	426.4
beecount	804.9	873.5	828.4	841.1
bw	3008.9	3289.6	3034.2	3067.0
clip	5329.6	6333.7	5551.6	5642.1
count	2613.2	3038.9	2722.0	2756.2
cse	2949.5	3627.9	3088.3	3167.4
dk14	1463.5	1653.2	1524.4	1534.8
donfile	1362.2	1620.4	1418.9	1427.2
duke2	8122.4	10516.7	8560.8	8758.6
esd	6437.9	8281.5	6756.1	6994.9
f51m	2210.5	2437.5	2282.6	2278.5
frg1	4520.2	5062.3	4708.5	4792.2
jay	4347.9	5627.9	4509.0	4599.2
keyb	5046.6	6047.1	5256.8	5359.5
irkman	2049.4	2343.9	2134.7	2167.2
lion	295.4	295.4	295.4	295.4
misex1	1043.5	1111.0	1066.9	1068.5
planet	7489.8	8983.6	7801.8	7958.1
platcos	5511.0	5694.0	5540.6	5549.5
rd53	1143.1	1173.4	1150.7	1164.4
rd84	11012.5	14498.1	12471.3	13200.5
s1a	3138.7	3814.7	3269.4	3328.5
sand	9677.8	11984.1	10081.0	10271.6
sao2	2614.6	3290.2	2743.5	2817.8
styr	6239.2	7854.2	6521.1	6650.9
tbk	4993.6	6487.6	5301.6	5457.0
vg2	4144.5	5187.8	4347.1	4439.2
z4ml	1532.6	1587.2	1536.4	1538.2
Gain comparatif	Ref.	16%	4%	6.2%

Tableau 3.2: Notre procédure d'isomorphisme pour l'optimisation en puissance.

méthode probabiliste délai zéro précédemment présentée. L'unité de puissance est le micro Watt par MHz. L'isomorphisme booléen est potentiellement meilleur que son équivalent structurel mais puisque sa mise en œuvre est plus complexe, les cellules complexes ne peuvent presque jamais être reconnues. La première heuristique, ici mauvaise, sera utilisée dans le cas d'une bibliothèque majoritairement composée de cellules complexes.

3.4 Sélection orientée puissance

3.4.1 Position du problème

Les modèles incorporables dans le processus de sélection n'optimisent que la puissance logique. Cela signifie que 20% de la puissance dynamique est ignorée. Mais cela peut aller beaucoup plus loin si le circuit à décomposer est très déséquilibré puisque les transitions parasites (*glitches*) sont sensibles aux différences de délai qui dépendent directement de la profondeur du circuit. Une deuxième source de génération de transitions parasites est la non-synchronisation des entrées du circuit combinatoire sur le même front d'horloge. En effet, l'application d'un tel vecteur d'entrées revient en fait à appliquer plusieurs vecteurs consécutifs temporellement corrélés. Par exemple, considérons le circuit de la figure 3.11, composé de deux *Xor* de temps de propagation nul (modèle délai zéro) mais dont les entrées sont synchronisées sur trois fronts successifs. La stimulation du vecteur proposé fournit 5 transitions alors que la synchronisation des entrées sur le même front d'horloge n'en fournit qu'une.

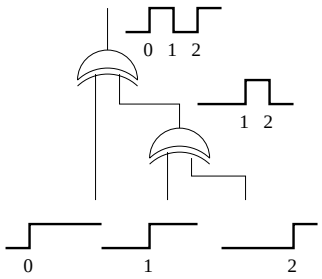

Figure 3.11: Synchronisation des entrées du circuit.

Dans ces deux conditions, le modèle délai zéro synchronisé sur le même front d'horloge s'avère être très éloigné de la réalité. Il serait donc utile de proposer une modélisation, même approchée, de la puissance totale, incorporable dans le processus de sélection, et dont le but serait de modifier la sélection des portes en fonction de la propagation réelle des transitions.

3.4.2 Modélisation des transitions inutiles

La commutation inutile de la sortie des cellules combinatoires dépend des deux facteurs suivants [31]: la **différence de temps d'arrivée** aux entrées de la porte et la **sensibilité** de celle-ci aux signaux d'entrée. La commutation est créée à la sortie d'une porte à cause de la différence des temps d'arrivée à ses entrées. Alors, la transition peut-être propagée aux portes de sortie suivant leurs sensibilités. La probabilité d'une telle commutation ne peut-être estimée de la même manière qu'une transition logique pour deux raisons: premièrement, la probabilité ne dépend plus seulement de la fonctionnalité du réseau booléen mais aussi de sa structure (la longueur des chemins par exemple). Deuxièmement, les vecteurs consécutifs sont maintenant corrélés. Par exemple, au temps t correspondant à une commutation possible, les entrées dont le temps d'arrivée est égal à t peuvent changer, les autres pas. Par conséquent, les deux vecteurs consécuctifs (à t et $t+1$) ne sont plus indépendants. La formule (2.4) n'est donc plus valide pour ces transitions.

Afin de résoudre ce problème, le calcul des probabilités est basé sur le délai de propagation et la sensibilité de chaque porte: les probabilités de commutation et les temps d'arrivée sont connus à l'entrée du circuit. Ils sont propagés à travers le circuit en utilisant le délai réel des portes. Chacune d'entre elles modifie la probabilité de commutation en fonction de sa capacité à propager la commutation des entrées à sa sortie, ce que nous appelons sensibilité. Finalement, les probabilités de toutes les commutations en sortie de portes sont additionnées pour fournir la puissance dynamique moyenne.

La première étape consiste donc à propager les temps d'arrivée des commutations à travers le circuit. Pour cela, le temps de propagation réel (2.1) est utilisé. Toutefois, deux transitions proches dans le temps à l'entrée ne fourniront pas deux transitions en sortie si l'intervalle de temps est plus petit que le temps de propagation de la cellule. Nous discrétisons donc les temps d'arrivée en sortie de la cellule de telle sorte qu'ils soient tous séparés d'au moins le temps moyen de propagation de la cellule. Ces propagations sont illustrées sur l'exemple 3.12 qui est la propagation en délai réel ou zéro de signaux synchronisés sur le même front d'horloge. Le modèle délai zéro ne fournit qu'une seule transition en sortie de la porte *Ou*, tandis que le délai réel en génère deux.

La seconde étape consiste à associer à ces commutations une probabilité dont la valeur dépend de celle de la commutation en entrée de la cellule, pondérée par la sensibilité de celle-ci. Reste donc à calculer la sensiblité d'une porte.

Definition 3.4.1 (Sensibilité d'une cellule) *La sensibilité d'une cellule caractérise sa capacité à propager les commutations des entrées vers la sortie.*

La sensibilité dépend de l'ensemble des entrées qui commutent ainsi que de l'état de toutes les entrées. Considérons à nouveau le circuit de la figure (3.11) pour s'en

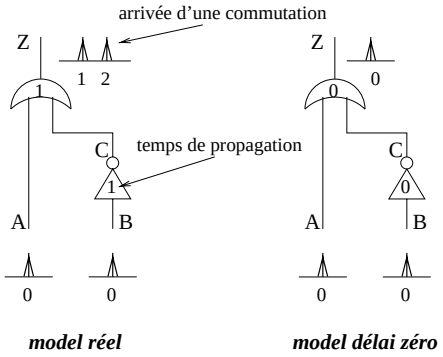

Figure 3.12: Propagation des temps d'arrivée.

assurer. En modèle délai zéro, les deux entrées du premier *Xor2* commutent en même temps donc la sortie ne commute pas. En revanche, en modèle réel, seule une des deux entrées commute donc la sortie commute. De plus dans le circuit de la figure (3.12), si l'entrée non inversée est à 0, la cellule *Ou2* se comporte comme un inverseur donc propage très bien les commutations de l'entrée complémentée. Par contre, si elle est à 1, la sortie ne peut jamais changer de valeur quelque soit le comportement de l'entrée complémentée.

Considérons une cellule de N entrées, dans un état probabiliste P, et subissant la commutation de l'ensemble des entrées S. On définit alors: c_i tous les vecteurs d'entrée possibles ($0 \leq i < 2^n$), $sw(c_i, S)$ le vecteur d'entrée obtenu en commutant dans c_i l'ensemble des entrées S, $g(c_i)$ la valeur de la sortie pour le vecteur c_i et $p(c_i)$ la probabilité d'occurrence du vecteur c_i. Alors la sensibilité moyenne de la cellule C est la somme des probabilités des vecteurs d'entrée pour lesquels la commutation de S founit une commutation en sortie:

Propriété 3.4.1 *La sensibilité moyenne de la cellule C, pour l'ensemble d'entrées S, dans un état probabiliste P est:*

$$Sen(C, S, P) = \sum_{0 \leq i < 2^n} p(c_i).[g(c_i) \oplus g(sw(c_i, S))]. \tag{3.8}$$

La probabilité de chaque vecteur d'entrée se calcule soit directement comme le produit des probabilités des entrées si celles-ci sont indépendantes ou en utilisant à nouveau les BDD dans le cas contraire. Cette sensibilité est moyenne puisqu'elle suppose que tous les états des entrées peuvent être atteints en fonction de la probabilité de chacune d'entre elles. En supposant une indépendance entre la sensibilité et la probabilité de commutation de S, alors

Propriété 3.4.2 *La probabilité moyenne de commutation en sortie de C pour la commutation de S, dans un état probabiliste P, est:*

$$P_{sw}(C, S, P) = P_{sw}(S).Sen(G, S, P). \tag{3.9}$$

Pour que cette probabilité soit moyenne, il faut considérer l'ensemble des combinaisons S possibles avec leur probabilité respective. Si deux entrées A et B peuvent commuter en même temps avec des probabilités $P_{sw}(A)$ et $P_{sw}(B)$, il faut en effet considérer les combinaisons AB, A et B avec les probabilités respectives $P_{sw}(A)P_{sw}(B)$, $P_{sw}(A)(1-P_{sw}(B))$ et $(1-P_{sw}(A))P_{sw}(B)$:

Propriété 3.4.3 *Pour les s entrées de même temps d'arrivée (avec $P^1 = P$ et $P^0 = 1-P$), on considère les ensembles S:*

$$\forall (i_1, i_2, ..., i_s) \in [0,1]^s, S = \{s_j / i_j = 1\} \text{ avec } P_{sw}(S) = P_{sw}^{i_1}(s_1)P_{sw}^{i_2}(s_2)...P_{sw}^{i_s}(s_s). \tag{3.10}$$

L'approximation de notre méthode vient de (3.9) puisque la simulation montre que la sensibilité de C n'est pas indépendante des entrées S, c'est-à-dire que les commutations des entrées S ne se répartissent par aléatoirement par rapport aux vecteurs d'entrée. A titre d'exemple, calculons le nombre de commutations en sorties des circuits de la figure (3.12):

- en modèle réel, l'ensemble des entrées commutantes sont {A} puis {C} avec une probabilité de $\frac{1}{2}$. La sensibilité est la somme des probabilités des vecteurs pour lesquels {C} puis {A} est à 0: $Sen(Ou2, \{A\}, \frac{1}{2}, \frac{1}{2}) = \frac{1}{4} + \frac{1}{4} = \frac{1}{2}$. La probabilité de commutation de Z est donc $P_{sw}(Z) = P_{sw}(A)\frac{1}{2} + P_{sw}(C)\frac{1}{2} = \frac{1}{2}$. Pour les 16 paires de stimuli possibles, on obtient 8 commutations de Z, ce qui correspond bien à $16 * \frac{1}{2}$.
- En modèle délai zéro par contre, les deux entrées A et C arrivent en même temps, donc les ensembles A, C et AC doivent être considérés (cf 3.10) avec les probabilités respectives $\frac{1}{2}(1-\frac{1}{2}) = \frac{1}{4}$, $(1-\frac{1}{2})\frac{1}{2} = \frac{1}{4}$ et $\frac{1}{2}\frac{1}{2} = \frac{1}{4}$. Les sensibilités correspondantes sont $Sen(Ou2, \{A\}, \frac{1}{2}, \frac{1}{2}) = \frac{1}{2}$, $Sen(Ou2, \{C\}, \frac{1}{2}, \frac{1}{2}) = \frac{1}{2}$ et $Sen(Ou2, \{A, C\}, \frac{1}{2}, \frac{1}{2}) = \frac{1}{2}$. En sommant ces trois possibilités, la probabilité de commutation en sortie est (eq 3.9): $P_{sw}(Z) = \frac{1}{8} + \frac{1}{8} + \frac{1}{8} = \frac{3}{8}$. La simulation des mêmes 16 paires de vecteurs d'entrée fournit à présent 6 commutations de sortie ce qui correspond bien à $16 * \frac{3}{8}$.

3.4.3 Résultats expérimentaux

Nous avons utilisé notre modélisation [32] dans deux expérimentations différentes: premièrement, nous comparons notre simulateur par rapport à la simulation exhaustive des circuits dans les cas d'une synchronisation sur le même front d'horloge ou sur plusieurs fronts. Le but de cette comparaison est de montrer que notre modélisation réalise de meilleures estimations que le modèle délai zéro. Deuxièmement, nous incorporons le modèle dans le processus de sélection afin de trouver une combinaison de cellules qui minimise la puissance totale et non plus la puissance logique.

La simulation exhaustive consiste à appliquer toutes les paires de stimuli possibles, afin de s'affranchir de la corrélation temporelle. Par exemple, un circuit élémentaire de deux entrées possède 4 stimuli possibles donc 16 paires de stimuli à appliquer. Nous nous limitons aux circuits pour lesquels le nombre d'entrées est faible (5 au plus) puisque la simulation exhaustive exige déjà 2^{10} stimuli. Nous recouvrons ces réseaux booléens par des portes simples (*Et*, *Ou* et *Inv*) et lançons les deux estimateurs (tableau 3.3). Le modèle délai zéro n'estime pas 13.5% de la puissance dynamique mais la moitié dans le cas de plusieurs fronts de synchronisation. Notre estimateur estime mieux la puissance dynamique bien qu'il s'en écarte lorsque la profondeur du réseau augmente. En effet, le calcul de puissance s'appuie sur les probabilités de commutations des portes d'entrées, donc peut perdre de la précision lorsque le nombre de portes consécutives est trop important. Toutefois, nous comparerons notre estimateur dans le cas d'architectures arithmétiques (additionneurs et multiplieurs) profondes, par rapport aux nombres théoriques de commutations. Pour ces circuits également, il s'avère beucoup plus précis que le modèle délai zéro. Sur les exemples testés, les transitions inutiles sont estimées à 12% près.

benchs	même front			fronts différents		
	simulation transitions	délai zéro erreur	délai réel erreur	simulation transitions	délai zéro erreur	délai réel erreur
bbtas	15102	-14%	1.9%	32451	-59.9%	8.7%
bw	107652	-12%	1.8%	257652	-63.2%	13.4%
dk15	34508	-14%	1.6%	61409	-51.6%	7.3%
dk17	30040	-14%	0.7%	55039	-53.0%	8.4%
dk27	3008	-13%	2.0%	5011	-47.7%	3.3%
lion	2220	-12%	0.9%	2980	-34.4%	4.2%
rd53	31600	-13%	0.9%	69301	-60.3%	9.7%

Tableau 3.3: Notre modélisation de puissance dynamique.

Nous avons incorporé notre modélisation dans le processus de programmation dynamique pour décomposer plusieurs circuits combinatoires (tableau 3.4.3). La sélection du meilleur isomorphisme s'effectue sur deux niveaux comme la vitesse puisque tous les isomorphismes n'ont plus la même dissipation de puissance. Notons que la programmation dynamique n'assure pas non plus le choix optimal puisque le choix d'un isomorphisme peut dépendre du choix des isomorphismes sur plusieurs niveaux en sortie. Les valeurs fournies sont celles estimées par un simulateur utilisant un ensemble important, mais pas exhaustif, de stimuli. Les circuits fournis en exemple ont, à présent, trop d'entrées. Toutefois, le nouveau coût de la programmation dynamique entraîne une optimisation de 3% de la puissance dynamique. Ce gain peut paraître faible mais les possibilités d'optimisation sont également faibles puisque la

puissance inutile à optimiser ne dépasse pas 20% de la puissance dynamique.

benchmarks	notre optimisation	optimisation délai zéro	gain (%)
9sym	55.3	65.0	14.9
alu4	463.7	473.9	2.1
apex1	663.3	673.4	1.4
apex6	269.6	274.2	1.6
apex7	108.8	114.9	5.3
dk14	45.7	46.6	1.9
donfile	90.4	94.8	4.6
esd	104.0	105.6	1.5
s1	125.4	128.7	2.5
styr	153.9	156.3	1.5
z4ml	23.9	24.6	2.8

Tableau 3.4: Sélection des isomorphismes

3.5 Conclusion

Nous avons présenté l'état de l'art des algorithmes de décomposition technologique et mis en valeur leurs faiblesses quant à une utilisation poussée des bibliothèques de cellules standards. Nous avons ainsi proposé d'améliorer le recouvrement en utilisant les cellules les plus complexes et en les sélectionnant judicieusement, afin de diminuer la puissance dynamique totale. Nous avons prouvé la validité d'une telle approche à l'aide du critère de la puissance dissipée, pariculièrement approprié aux méthodes de réutilisation.

En combinant la procédure d'isomorphisme de portes complexes et la procédure de sélection pour la puissance dynamique, nous optimisons le recouvrement orienté puissance de 5% à 10% comparé aux algorithmes classiques de décomposition technologique. Notons toutefois que nous nous sommes concentrés sur des aspects particuliers de la puissance (portes complexes et transitions inutiles). Comme pourra le montrer la chapitre suivant, l'optimisation peut être beaucoup plus importante au niveau architectural.

Les travaux futurs d'isomorphisme sont susceptibles de s'orienter dans deux axes: d'abord améliorer les techniques d'isomorphismes booléens afin de mieux utiliser les portes complexes de la bibliothèque, puisque l'approche booléenne apporte un meilleur formalisme mathématique et de meilleures perspectives pour l'étape de décomposition technologique. Nous pouvons citer, notammment, les travaux de De Micheli de

l'université de Stanford. D'un autre côté, il est également possible d'adapter la synthèse physique et la génération automatique du dessin des masques à la technologie cible. De cette façon, la synthèse physique doit produire une réalisation paramétrable et optimisée des réseaux de portes. Nous pouvons ici citer les travaux effectués au LIRMM sous la direction de Michel Robert sur la *Synthèse automatique de macrocellules*.

Chapitre 4

Conception des Blocs Arithmétiques

4.1 Introduction

La synthèse RTL, ou transfert de registres, optimise les opérations dans une période d'horloge. Celle-ci transforme un circuit spécifié pour chaque cycle d'horloge en un ensemble d'équations booléennes. On distingue l'élaboration et la macro-génération: tandis que l'élaboration compile le code source en un modèle interne où sont spécifiés l'ensemble des registres, des opérateurs et les transferts entre registres et opérateurs, la macro-génération transforme les machines et opérateurs en équations: les opérateurs, arithmétiques par exemple, sont transformés en équations contenues dans les éléments de bibliothèque. La figure 4.1 montre comment le macro-générateur utilise les fonctions arithmétiques contenues dans la bibliothèque pour recouvrir un opérateur HDL. La bonne architecture doit être sélectionnée en fonction des contraintes de la synthèse. Dans l'exemple, un additionneur rapide est choisi.

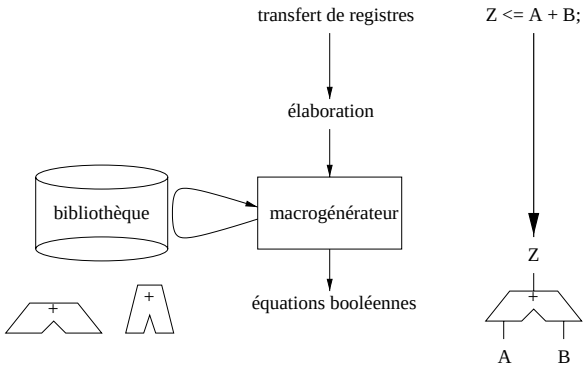

Figure 4.1: L'étape de réutilisation en synthèse RTL.

Le point important de ce niveau est le petit nombre de blocs réutilisables. Il s'agit dans la très grande majorité des additionneurs et multiplieurs (et de leurs circuits dérivés), très rarement la division. Par conséquent, l'objectif d'un macro-générateur est de proposer ces deux blocs de telle manière qu'ils satisfassent toutes les exigences du concepteur. Ces contraintes sont de deux types:

- Les **contraintes temporelles** dans une synthèse dirigée par la vitesse. Il s'agit de proposer le bloc le plus petit possible, tout en répondant à toutes les contraintes temporelles. Par exemple, des outils commerciaux itèrent sur l'ensemble des architectures qu'ils possèdent, par ordre de surface croissante, jusqu'à celle satisfaisante. Nous proposons de définir l'architecture interne du bloc arithmétique dynamiquement pour chaque jeu de contraintes afin d'en minimiser la surface.

- Les **contraintes technologiques**. Puisque ce macro-générateur peut-être utilisé pour des cibles telles que les FPGA ou CPLD, qui représentent des contraintes technologiques très fortes, nous ne pouvons nous contenter de décomposer des équations sur des modules programmables. Celles-ci ne seront qu'insuffisamment exploitées et les réseaux programmables seront très insaturés. Pour les cibles Xilinx et AMD, nous proposons des architectures spécifiques afin d'optimiser le critère vitesse.

La construction d'un macro-générateur est la possibilité que nous avons pour mettre en œuvre l'approche réutilisation de la synthèse RTL. Nous utiliserons les estimateurs développés au chapitre précédent afin de caractériser la puissance dissipée des blocs arithmétiques et de définir, pour les plus simples d'entre eux, une consommation totale analytique. On fournira également les meilleurs compromis vitesse-puissance des multiplieurs. Il est important de comprendre qu'il ne s'agit pas de construire une bibliothèque figée de blocs arithmétiques. On ne disposerait que d'un petit nombre d'architectures qui n'auraient que peu de chance de satisfaire à toutes les contraintes du circuit. Il s'agit ici de concevoir l'architecture du bloc au cours de la synthèse, de telle sorte qu'une architecture spécifique soit construite pour chaque jeu de contraintes spécifiques. Il devient donc possible de mixer plusieurs algorithmes classiques pour obtenir la solution adéquate.

Le chapitre est composé comme suit: suite à une présentation de l'état de l'art, nous proposons un formalisme du parallélisme de l'addition dans la seconde section. Appliqué aux contraintes temporelles et technologiques, il produit des additionneurs rapides sur ASIC (section 4) et FPGA/CPLD (section 5). De même dans les sections 6 et 7, la multiplication sera implantée sur les deux mêmes cibles.

4.2 Les additionneurs classiques

4.2.1 Principe de l'addition

Les circuits arithmétiques utilisent des entiers dont la position des bits est numérotée de 0 à $n-1$, le bit de poids le plus faible étant à droite des figures. A chaque position i, la retenue sortante c_{i+1} peut-être soit **générée** ($c_{i+1}=1$), soit **détruite** ($c_{i+1}=0$), soit **propagée** ($c_{i+1}=c_i$). Puisque seuls deux de ces signaux sont nécessaires, nous retiendrons la génération ($g_i = a_i.b_i$) et la propagation ($p_i = a_i \oplus b_i$) pour la position i. L'addition consiste donc à calculer les signaux (p_i, g_i) pour $i \in [0, n-1]$, à en déduire les retenues c_i pour $i \in [1, n-1]$, puis à déterminer les sorties $s_i = p_i \oplus c_i$ pour $i \in [0, n-1]$. Des méthodes directes permettent de réaliser l'addition sans l'utilisation des termes de propagation et génération, mais on peut toujours les modéliser grâce à ce formalisme.

Alors que les signaux (p_i, g_i) sont définis pour la position i, on cherche à définir les signaux (P_i^j, G_i^j), notés PG_i^j pour la tranche de bits $[j, i]$. P_i^j signifie que la retenue se propage de la position j à la position i ($c_{i+1} = c_j$) et G_i^j signifie que la retenue est générée quelque part entre j et i et propagée jusqu'à la position i ($c_{i+1} = 1$). Pour ce faire, on utilise les groupes consécutifs PG_{k-1}^j et PG_i^k selon la relation:

$$PG_i^j = (P_i^k.P_{k-1}^j, G_i^k + P_i^k.G_{k-1}^j). \tag{4.1}$$

Puisqu'en absence de retenue entrante, $c_i = G_{i-1}^0$, (4.1) permet de calculer les sorties en fonctions des retenues c_i. (On ne calculera jamais les termes P_{i-1}^0.)

4.2.2 Organisations classiques

Les organisations classiques ne représentent que la structure de calcul des retenues c_i. On ne représente pas les étapes initiale et finale. Ils s'articulent autour de la cellule Δ, réalisant la relation (4.1), comportant donc quatre entrées et deux sorties selon la figure 4.2. Il ne s'agit pas d'une cellule d'addition *full adder*. Bien entendu, ces équations seront convenablement décomposées sur des cellules rapides inverseuses de la bibliothèque cible dans la phase de décomposition technologique, mais puisque tous les additionneurs seront équitablement décomposés, le nombre et la profondeur de cellules Δ est une excellente mesure de la complexité et du délai de ces architectures.

L'additionneur *ripple carry* consiste à calculer **sériellement** les retenues. On peut également intégrer la cellule Δ avec les étages initiaux et finaux dans un *full adder*. La surface et le délai de cette architecture sont **linéaires** par rapport à la taille de l'addition. Si on utilise des portions de *ripple carry* de tailles croissantes en les connectant astucieusement, on obtient un additionneur *carry select*. La complexité double mais le délai est en racine carrée de la longueur du mot. La figure 4.3 montre bien comment ces deux architectures ont en commun l'utilisation de portions de cellules Δ connectées en série. Alors que l'architecture de gauche est entièrement

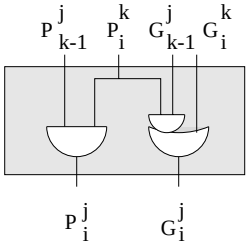

Figure 4.2: Cellule Δ.

sérielle, celle de droite ne comporte que de petites portions d'additions en série de tailles croissantes. Elles sont donc petites mais très lentes.

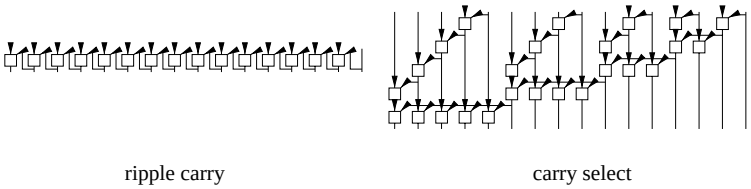

Figure 4.3: Additionneurs séries *ripple carry* et *carry select*.

La seconde grande catégorie (additionneurs *carry look-ahead*) consiste à traiter les positions **parallèlement** pour un accroissement de la complexité mais pour un délai **logarithmique**. Pour tous, un arbre binaire est construit pour les positions en puissance de deux. Les différentes architectures se distinguent pour les sorties restantes: l'additionneur de Kogge et Stone [33] copie l'arbre binaire pour les bits de poids plus faible. Sklanski construit également un arbre logarithmique mais utilise celui de la position en puissance de deux précédentes [34]. La complexité est donc plus faible mais la sortance croît exponentiellement. Ces deux solutions sont structurellement optimales. Brent et Kung construisent le reste des positions en utilisant les bits les plus significatifs déjà construits [35]. Finalement, l'additionneur de Han et Carlson [36] utilise l'architecture de Kogge et Stone pour les bits pairs et déduit les positions impaires en une couche supplémentaire. La figure 4.4 résume les 4 algorithmes classiques *carry-look-ahead*: ils ont en commun une profondeur logarithmique, c'est-à-dire un nombre de cellules correspondant au logarithme de la taille des opérandes. Pour les additionneurs de 16 bits présentés, l'architecture de *Sklanski* ne totalise que $log(16) = 4$ cellules sur le chemin critique, contre 15 pour l'additionneur *ripple carry*.

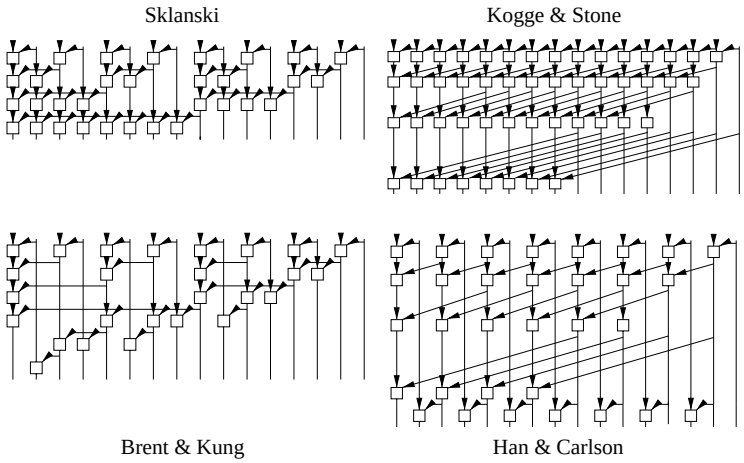

Figure 4.4: Additionneurs *carry look-ahead*.

4.2.3 Modèles de puissance

L'activité causée par une propagation de retenue sur k positions d'un additionneur *ripple carry* est proportionnel à $\frac{k^2}{2}$ selon [37], la propagation s'effectuant sur une chaine de bits à 1 consécutifs. Or le nombre de chaines de 1 de longueur k dans un mot de longueur n est $N(n,k) = 2^{n-k}(1 + \frac{n-k-1}{4})$ [38]. On en déduit l'activité moyenne:

$$A_{totale}^{ripple} = \frac{1}{2^n}\sum_{k=0}^{n}\frac{k^2}{2}N(n,k) = \frac{3n-4}{4} - \frac{2n^2}{2^{n+3}} \longrightarrow \frac{3n}{4}. \tag{4.2}$$

Puisque les sorties sont équiprobables, l'activité logique moyenne est:

$$A_{logique}^{ripple} = \frac{n}{2}. \tag{4.3}$$

On en déduit un rapport entre puissance inutile et logique de $\frac{1}{3}$. Pour le *carry select*, le premier niveau consiste en des *ripple carry* de taille croissante (de 1 à $\sqrt{n}$), et le deuxième niveau est approché par un *ripple carry* mais dont la capacité attaquée à la position i est proportionnel à i. L'activité totale ainsi comptabilisée s'élève à:

$$A_{totale}^{select} \longrightarrow \frac{n\sqrt{2n}}{2}, \tag{4.4}$$

pour une activité logique de:

$$A_{logique}^{select} = n - \frac{\sqrt{2n}}{2}. \tag{4.5}$$

Les additionneurs *carry look-ahead* n'ont pas, ou très peu de puissance inutile puisque tous les chemins sont de même longueur. Le terme de propagation *Et*

s'atténue très vite alors que le terme de génération *Ou* reste quasiment constant à $\frac{1}{2}$. Cette approximation permet d'évaluer la puissance totale de l'additioneur Kogge et Stone à la moitié du nombre de cellules, soit:

$$A_{totale}^{KS} = \frac{n}{2}.log_2(n). \tag{4.6}$$

Les activités théoriques que nous avons ici calculées sont vérifiées grâce à l'estimateur développé dans le chapitre précédent. Cette comparaison est une validation supplémentaire de notre estimateur sur des circuits de grande profondeur. Nous fournirons une caractérisation expérimentale de ces puissances dissipées dans la section consacrée aux multiplieurs.

4.2.4 Comparaisons

Les architectures classiques d'additionneurs sont récapitulées dans le tableau 4.2.4 [39], où figurent la surface et le délai en terme de cellules Δ, la sortance maximale ainsi que la puissance moyenne totale et logique. Les délais sont linéaires ou logarithmiques, la surface linéaire ou fonction du produit $n.log_2 n$. La sortance est en général faible mais linéaire pour l'additionneur de *Sklanski*, ce qui le pénalise pour des opérandes de grandes tailles. En effet, dans une technologie sensible à la sortance, une sortance de quatre ou cinq équivaut au délai d'une porte logique. Enfin, la puissance dissipée est proportionnelle à la surface, mais présente un portion significative de transitions redondantes pour les additionneurs séries.

Architectures	surface	délai	sortance maximale	activité totale	activité logique
ripple carry	n - 1	n - 1	2	$\frac{3n}{4}$	$\frac{n}{2}$
carry select	$\lceil 2n - \sqrt{2n} \rceil$	$\lceil \sqrt{2n} \rceil$	$\lceil \sqrt{2n} \rceil$	$\frac{n\sqrt{2n}}{2}$	$\frac{\lceil 2n - \sqrt{2n} \rceil}{2}$
Sklanski	$\lceil \frac{n}{2} log_2 n \rceil$	$\lceil log_2 n \rceil$	$\frac{n}{2}$	-	-
Kogge-Stone	$\lceil n(log_2 n - 1) \rceil$	$\lceil log_2 n \rceil$	2	$\frac{n}{2}.log_2 n$	$\frac{n}{2}.log_2 n$
Brent-Kung	$\lceil 2n - log_2 n \rceil$	$\lceil 2log_2 n - 2 \rceil$	$\lceil 2log_2 n - 2 \rceil$	-	-
Han-Carlson	$\lceil \frac{n}{2} log_2 n \rceil$	$\lceil log_2 n \rceil + 1$	2	$\approx \frac{n}{4}.log_2 n$	$\approx \frac{n}{4}.log_2 n$

Tableau 4.1: Comparaison des architectures classiques d'additionneurs.

4.3 Formalisation du parallélisme de l'addition

La présentation précédente des additionneurs classiques a mis en évidence leurs architectures sérielles ou parallèles. Nous proposons donc un formalisme exprimant le degré de parallélisme de la propagation de la retenue. De cette modélisation, on en déduit des techniques d'exploration de l'espace des solutions dont le but est d'obtenir la

meilleure solution pour chacun des circuits synthétisés. Nous présenterons également des méthodes applicables aux technologies FPGA/CPLD et aux contraintes temporelles spécifiques.

4.3.1 Tranche d'addition

On considère une tranche de bits compris entre les bits j et i dans laquelle se propage une retenue.

Definition 4.3.1 *La tranche $\Delta[i,j]$ suivante est appelée Δtranche:*

$$\Delta[i,j] = PG_j^j, PG_{j+1}^j, \cdots PG_i^j.$$

Une Δtranche est une addition effectuée sur une portion du mot (entre j et i) pour laquelle tous les termes de génération et propagation PG_k^j ($j \leq k \leq i$) sont calculés. Toutefois, l'architecture de cette tranche n'est pas spécifiée. Le but de l'addition est de calculer la tranche $\Delta[n-1,0]$. Pour atteindre cette tranche, des Δtranches plus petites peuvente être calculées et additionnées selon (4.1).

Definition 4.3.2 *Un Δarbre est un arbre dont les feuilles et les nœuds sont des Δtranches. Un nœud $\Delta[i,j]$ a pour sous-arbres les Δtranches $\Delta[k,j]$, $\Delta[l,k+1]$, $\cdots$, $\Delta[i,r+1]$ avec $j \leq k \leq l \cdots r \leq i$, si et seulement si $\Delta[i,j]$ est calculée à partir des Δtranches précédentes. Les bits $k,l,\cdots r$ sont appelés bits de coupe.*

L'addition des Δtranches s'effectue comme si celles-ci étaient de simples bits, à la différence que tous les bits d'une tranche doivent avoir leur terme de propagation et de génération calculés. Les additionneurs classiques s'expriment donc par un Δarbre plus ou moins profond dont les tranches peuvent être réalisées sériellement ou parallèlement. On peut forcer les Δtranches à être uniquement serielles: le Δarbre est donc unique. Toutefois, certaines architectures ne peuvent pas être représentées par un tel arbre. Tandis que si l'architecture des Δtranches n'est pas contrainte, tous les additionneurs peuvent être décrit de la sorte.

Dans ce qui suit, nous donnons la représentation en Δarbre la plus fine possible, puisqu'en concaténant les Δtranches consécutives, on peut trouver beaucoup de Δarbres équivalents. Comme le montre la figure 4.5, un additionneur *ripple carry* est une concaténation sérielle de Δtranches élémentaires, un *carry select* est une concaténation sérielle de Δtranches sérielles de longueurs croissantes, et un *carry look-ahead Sklanski* est un arbre logarithmique de Δtranches élémentaires.

Les autres additionneurs parallèles (à délai logarithmique) ne se représentent pas par un Δarbre de Δtranches sérielles puisque les pseudo-Δtranches consécutives contiennent certains bits sans terme de propagation ou de génération (*Brent-Kung* ou *Han-Carlson*) ou que les sous Δtranches à concaténer ne sont pas consécutives (*Kogge-Stone*). Toutefois, l'arbre $\Delta[15,0]$ les représente sachant que la structure de cette tranche est une architecture particulière (BK par exemple). Cela permet d'utiliser, dans la construction d'un Δarbre, des Δtranches de toutes les architectures possibles.

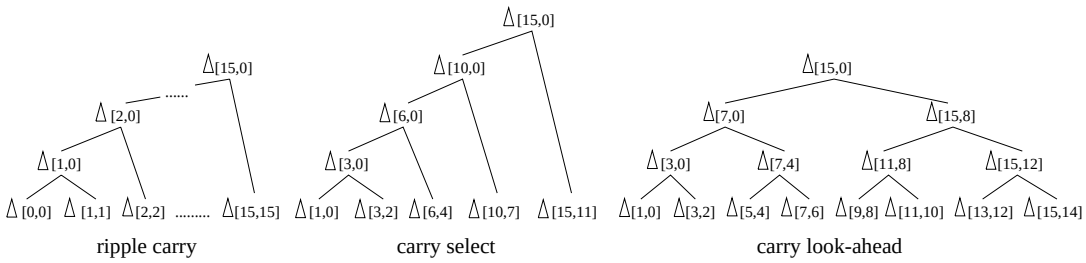

Figure 4.5: Δarbre des additionneurs classiques.

4.3.2 Exploration de l'espace des solutions

Puisque beaucoup d'additonneurs s'expriment par un Δarbre de Δtranches sérielles, on peut en modifier la structure pour parcourir une partie de l'espace des solutions [40]. En partant d'un Δarbre existant, on modife sa structure en **créant**, **éliminant** ou **translatant** une coupe. La figure 4.6 gauche montre la translation du bit de coupe 2 au bit de coupe 3. Les deux tranches correspondantes sont donc décalées, de $\Delta[5,3]$ à $\Delta[5,4]$ par exemple. L'arbre de droite montre la création du bit de coupe 1, formant ainsi deux nouvelles Δtranches $\Delta[1,0]$ et $\Delta[3,2]$.

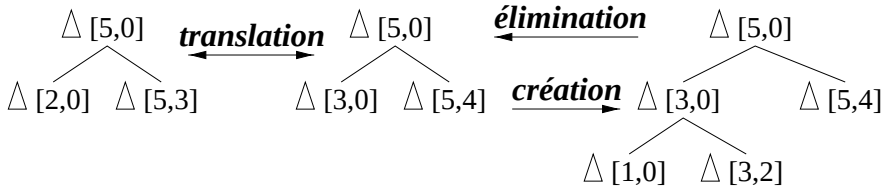

Figure 4.6: Modification de la structure d'un Δarbre.

La donnée d'un Δarbre est suffisante pour calculer son délai et sa surface puisque le délai d'une cellule Δ, la profondeur du Δarbre et la sortance sont connues. On peut donc parcourir beaucoup de solutions en très peu de temps (l'application d'une des trois opérations élémentaires modifie peu le calcul des performances de la solution précédente.) Le choix de l'opération dépend du critère de sélection: la création d'un point de coupe parallélise l'addition donc l'accélère et le grossit, l'élimination d'un bit de coupe le sérialise donc le ralentit et le densifie, tandis que la translation d'un bit de coupe peut avoir des effets variables suivant la taille des sous-arbres correspondants. Cette méthode itérative ne s'applique que lorsqu'il n'y a pas de contraintes fortes du circuit (temporelles ou technologiques) sans quoi les approches des deux sections suivantes seront appliquées.

Le tableau 4.2 représente les résultats de l'algorithme d'optimisation appliqué à

un arbre d'une bonne solution, à savoir celle générée par notre outil de synthèse lors d'une compilation en vitesse sous contrainte de surface. L'objectif est donc de diminuer le chemin critique sans augmenter significativement la complexité. Les gains moyens, obtenus pour la technologie *Compass vsc370* sont de 7.5% en performance pour une même surface moyenne. Notons que s'il est possible de diminuer le chemin critique en augmentant la surface, il est plus difficile de ne pas la détériorer. Lorsque les contraintes temporelles ou technologiques sont fortes et précises, la structure du Δarbre n'est plus déterminée de façon **aléatoire**.

Taille	additionneur initial		additionneur final		gain (%)	
	surface	CC	surface	CC	surface	CC
14	198	7.8	210	7.3	-6.0	6.4
16	258	8.4	244	7.8	5.4	7.1
18	281	8.5	281	8.1	0	4.7
20	283	9.3	302	8.8	-6.7	5.4
22	348	9.5	349	8.8	-0.3	7.4
24	310	10.0	304	9.7	0.2	3.0
26	381	10.3	381	10.3	0	0
28	514	10.1	502	9.2	2.3	8.0
30	567	11.0	574	9.5	-1.2	13.6
32	639	11.4	678	10.1	-6.1	11.4
34	788	11.1	727	10.3	7.7	7.2
total					-0.4	7.4

CC = chemin critique

Tableau 4.2: Optimisation aléatoire d'un Δarbre inital.

4.4 Optimisation de l'addition sous contraintes temporelles

4.4.1 Utilisation des contraintes temporelles

Le choix du Δarbre de l'additionneur est maintenant sélectionné en fonction des contraintes temporelles du circuit. Puisque l'additionneur est en fait immergé dans de la logique ou d'autres macro-blocs, les entrées ont des temps d'arrivées différents. Considérons par exemple la figure 4.7 où deux entiers de huit bits sont à additionner, pour lesquels les bits trois et sept sont en retard. (Les étages initiaux et finaux (Xor) ne sont pas représentés puisqu'ils rajoutent un délai constant.) Un additionneur *carry look-ahead* exige $\lceil log_2(8) \rceil = 3$ niveaux de cellules Δ donc le délai de l'additionneur est celui des trois cellules Δ. Par contre, un additionneur *ripple carry* profite du

retard du bit de poids fort pour effectuer tout le reste de l'addition. Une seule cellule Δ est alors suffisante pour finir l'addition. Dans ce cas, le délai de l'addition est celui d'une cellule.

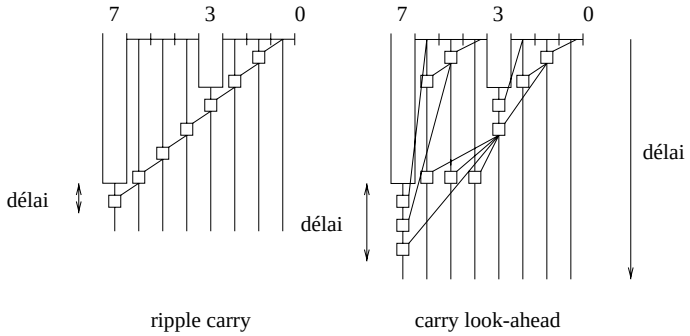

Figure 4.7: Un exemple d'utilisation des contraintes temporelles.

4.4.2 Motifs série et parallèle

L'utilisation d'une Δtranche sérielle ou parallèle ne dépend plus de leur structure mais de leur comportement en fonction des différents stimuli temporels appliqués. Un additionneur série n'est donc pas forcément le plus lent. Le schéma 4.8 résume le délai de chacune des architectures: dans le deuxième cas, le délai de la Δtranche parallèle **s'ajoute** au maximum des temps d'arrivées aux entrées de l'additionneur. La Δtranche est rapide mais supporte mal le retard d'une des entrées puisqu'elles sont toutes traitées en parallèle. On l'utilisera donc lorsque toutes les entrées sont alignées dans le temps. Dans le premier cas en revanche, les temps d'arrivées ne s'ajoutent plus au délai des cellules Δ mais **rivalisent** avec la retenue de la cellule précédente. La Δtranche sérielle convient parfaitement aux entrées dont les temps d'arrivée croissent progressivement.

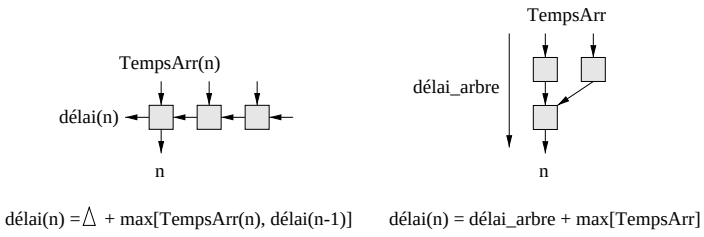

Figure 4.8: Motifs série et parallèle.

Les délais des Δtranches sérielles et parallèles sont respectivement:

$$\begin{cases} t_{série} & = max_{i=1}^{n-1}[i\Delta + TempsArr(n-i)] \\ t_{parallèle} & = \lceil log_2(n) \rceil \Delta + max_{i=0}^{n-1}[TempsArr(i)] \end{cases} \quad (4.7)$$

En utilisant l'équation (2.1) de propagation, ces équations deviennent:

$$\begin{cases} t_{série} & = max_{i=1}^{n-1}[i(\alpha + \beta C) + TempsArr(n-i)] \\ t_{parallèle} & = \lceil log_2(n) \rceil (\alpha + 2\beta C) - 3\beta C + max_{i=0}^{n-1}[TempsArr(i)] \end{cases} \quad (4.8)$$

avec $TempsArr(1) = max[TempsArr(1), TempsArr(0)]$ et où l'additionneur utilisé est l'architecture de Kogge et Stone. Pratiquement, nous utiliserons l'architecture de Han et Carlson puisqu'elle est quasiment optimale pour une complexité réduite de moitié.

4.4.3 Sélection du Δarbre

Le principe de notre algorithme consiste à traiter en premier les entrées **précoces** de telle sorte qu'une Δtranche peut-être formée avant l'arrivée des autres entrées. La limite des Δtranches correspond donc aux entrées les plus tardives. La conséquence est que le nombre de tranches à ajouter par la suite est réduit si bien qu'un additionneur plus petit est suffisant pour finir l'addition. Supposons que l'entrée $\frac{n}{2}$ soit tardive: la figure 4.9 montre que les groupes d'entrées $(0, \frac{n}{2}-1)$ et $(\frac{n}{2}+1, n-1)$ peuvent être additionnés pendant ce temps, c'est-à-dire que le délai des deux additionneurs n'excède pas le temps d'arrivée de l'entrée $\frac{n}{2}$. Ensuite, il ne reste plus que trois tranches à ajouter, de temps d'arrivée semblables. Par conséquent, un additionneur de trois bits suffit pour terminer l'addition.

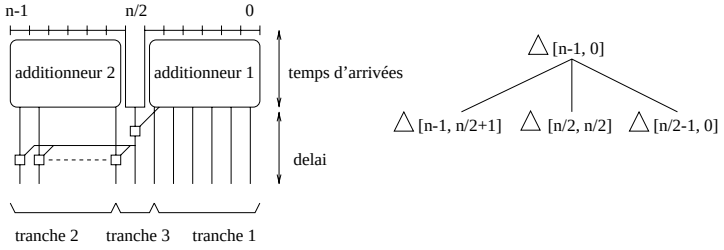

Figure 4.9: Principe de sélection du Δarbre.

Un problème équivalent consiste à traiter les entrées les plus **tardives** en dernier. La différence est que des Δtranches internes peuvent ne pas être nécessairement formées si le retard de ces entrées est suffisant. On prend pour contrainte t l'entrée plus tardive. On essaie de former les Δtranches dont les limites sont cette (ou ces) entrée(s) et dont le délai est au plus t. Si l'un des deux motifs convient pour toutes ces tranches, on suppose que celles-ci ont même délai que t. D'après ce qui précède, un

additionneur parallèle est particulièrement approprié pour terminer l'addition. Pour les tranches dont les deux motifs ont un délai trop grand, deux cas se présentent:

1. une des entrées au moins de la tranche est tardive. Cette entrée devient la nouvelle contrainte t' pour les autres entrées de la tranche et le même processus est récursivement appliqué. Toutes les entrées (ou Δtranches) ont à présent même temps d'arrivée t' et le deuxième cas peut-être appliqué.

2. toutes les entrées (ou Δtranches) ont même temps d'arrivée, donc le moins d'additionneurs possibles consécutifs (forcément parallèles) doivent être formés sous la contrainte t.

Cette méthode peut créer des sortances élevées (comme le *carry select*). Cet accroissement de délai doit être pris en compte dans les équations (4.8) lors de la construction des Δtranches. De plus, les larges sortances peuvent être optimalement réparties sur deux niveaux: le premier niveau de sortance $\lfloor\sqrt{(n)}\rfloor$ et le second de $\lceil\frac{n}{\lfloor\sqrt{(n)}\rfloor}\rceil$. L'algorithme construit récursivement des additionneurs dans un environnement temporel spécifique [41]. La complexité de l'algorithme est quasiment celle d'un additionneur *carry look-ahead* classique: il doit néanmoins tester de nombreux petits additionneurs (cas 2a/2b) mais ceci est effectué avant la phase de décomposition.

Algorithme 1 - programme principal

1. soit $[Ptard]_{i=1}^{p}$ les entrées tardives dans $[0, n-1]$ de délai t_{const}.
2. si ($[Ptard]_{i=1}^{p} \neq 0$), SynthAdd (0, $[Ptard]_{i=1}^{p}$, $n-1$, t_{const}).
3. SynthLeMoinsAddPossibleSousContrainte (0, n-1, dès-que-possible).

Algorithme 2 - SynthAdd (n_{min}, $Ptard_{i=1}^{p}$, n_{max}, t_{const})

1. pour chaque tranche $[N_1, N_2]$ dans ($[n_{min}, Ptard_1 - 1]$, $[Ptard_1 + 1, Ptard_2\text{-}1]$, ..., $[Ptard_p + 1, n_{max}]$)
2. faire
   (a) essayer le motif série dans $[N_1, N_2]$ sous t_{const}. si succès, construire l'additionneur et aller à la boucle suivante.
   (b) essayer le motif parallèle dans $[N_1, N_2]$ sous t_{const}. si succès, construire l'additionneur et aller à la boucle suivante.

(c) soit $[NPtard]_{i=1}^{q}$ les entrées tardives dans $[N_1, N_2]$ de délai t'_{const}

(d) si ($[NPtard]_{i=1}^{q} \neq 0$), SynthAdd ($N_1$, $[NPtard]_{i=1}^{q}$, N_2, $t'const$). (cas 1)

(e) SynthLeMoinsAddPossibleSousContrainte (N_1, N_2, t_{const}). (cas 2)

3. fin

4.4.4 Résultats expérimentaux

La conception d'additionneurs rapides a deux buts: d'une part d'autres macro-blocs utilisent intensivement les additions tels que les multiplieurs et diviseurs; d'autre part, les additionneurs sont souvent inférés dans des circuits plus importants. Nous comparons donc notre algorithme à un classique *carry look-ahead* pour la synthèse d'additionneurs dans les multiplieurs de Booth et dans les diviseurs (tableau 4.3), puis dans des circuits plus complets (tableau 4.4). Il s'agit donc de réaliser l'addition finale, en prenant en compte des temps d'arrivée différents. La technologie utilisée est la bibliothèque sub-micronique ($0.8\mu m$) *TSBC4* de *Thomson CSF semi-conducteurs spécifiques*. Dans les multiplieurs, les bits de poids faibles sont générés en premier tandis que dans les diviseurs, les poids forts sont générés en premier. Tous les macro-blocs sont optimisés grâce à notre algorithme sauf le multiplieur de 16 bits. Dans ce cas, les temps d'arrivées ne sont pas assez déséquilibrés pour éliminer un niveau de cellule Δ tandis que la création de sortance pénalise un peu le chemin critique. La division convient parfaitement à notre approche puisque des gains de chemin critique de l'ordre de 30% sont atteints pour les grandes tailles d'opérandes.

macro-bloc	taille addition	délai (ns)			surface (unités *TSBC4*)		
		CLA	algo 1	gains (%)	CLA	algo 1	gains (%)
multiplieur 8 bits	16	4.83	3.94	16.3	87962	78112	11.1
multiplieur 16 bits	32	5.40	5.69	-5.5	180575	167169	7.4
multiplieur 24 bits	48	6.00	5.77	3.8	295487	299591	-1.4
diviseur 8 bits	8	3.51	2.70	23.0	35715	40629	-14.0
diviseur 16 bits	16	4.49	3.46	23.0	77702	102736	-32.0
diviseur 32 bits	32	5.54	3.64	34.0	180575	232833	-29.0

CLA = *carry look-ahead*

Tableau 4.3: Additionneurs dans des macro-blocs complexes.

Pour l'utilisation d'additionneurs dans des circuits complets, nous avons sélectionnés quelques benchmarks comportant des additions pour lesquels les temps d'arrivées sont déséquilibrés. Dans la troisième colonne, nous avons écrit le type de logique ou de blocs, synthétisés avec l'outil ASYL+, et contenu en entrée de l'additionneur à optimiser. La taille des additionneurs est petite mais nous avons obtenu **les mêmes**

performances en étendant la description VHDL. Des gains jusqu'à 37% et 26% pour le délai et la surface sont atteints. Les gains en surface viennent du fait qu'une partie de l'addition est réalisée avec des additionneurs séries. Dans l'exemple *prep5-1*, les multiplieurs synthétisés fournissent des temps d'arrivée, en sortie de l'additionneur final, proportionnels au poids de la sortie. C'est-à-dire que les poids faibles sont produits en premier, et les poids forts en dernier. Il s'agit donc d'une condition particulièrement favorable à l'utilisation d'une architecture *ripple carry*.

bench	taille addition	logique en entrée	délai (ns)			surface (unités *TSBC4*)		
			CLA	algo 1	gains	CLA	algo 1	gains
arith3	8	additionneurs	3.51	3.41	2.9%	35715	33105	7.4%
add4	8	mux	3.51	3.16	10%	35715	33105	7.3%
prep5-1	8	multiplieur Braun, accu	3.51	2.17	38%	35715	24760	31.0%
prep5-1	8	multiplieur Wallace, accu	3.51	2.21	37%	35715	26265	26.0%
ex10	8	additionneurs mux	3.51	2.92	17%	35715	26265	26.0%
arithexp	8	additionneurs	3.51	2.99	15%	35715	27770	22.0%

Tableau 4.4: Additionneurs dans des circuits complets.

4.5 Optimisation de l'addition sous contraintes technologiques

La technologie cible doit également diriger la construction du Δarbre et la nature des Δtranches. Parce que la granularité des circuits FPGA et CPLD permet d'absorber plus de logique que les cellules standards et parce que ceux-ci proposent des avantages technologiques importants, le meilleur additionneur doit adapter sa structure aux contraintes suivantes.

4.5.1 Présentation des contraintes technologiques

Dans les technologies FPGA à base de *LUT* (*look-up table*), la série *Xilinx 5200* propose deux modules réalisant les fonctions combinatoires: d'un côté, les modules *FMAP* sont des *LUT* à quatre entrées pouvant réaliser n'importe quelle fonction de quatre variables au plus. De l'autre côté, les modules *CY-MUX* se comportent comme un multiplexeur [42], propageant une retenue. Dans une première approximation, le délai d'un *CY-MUX* est huit fois plus rapide que celui des *FMAP*. La logique contenue

dans un *CLB* est illustrée dans la figure 4.10.

Les CPLD proposent par ailleurs une macro-cellule réalisant n'importe quelle somme de monômes avec toutefois un nombre maximal de variables et de monômes. De la société *AMD*, le*Mach 5* permet de réunir 32 variables et 16 monômes dans un niveau de macro-cellule. Un schéma d'une telle macro-cellule est également fourni.

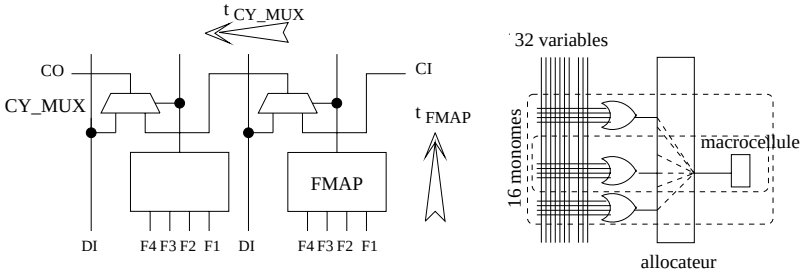

Figure 4.10: Blocs combinatoires des FPGA et CPLD.

4.5.2 Optimisation pour le FPGA Xilinx 5200

L'implantation sérielle ne requiert que les groupes de génération G_k^0 pour $0 \leq k \leq n-1$ selon: $G_k^0 = G_k^k + G_{k-1}^0.P_k^k$. Mais puisque $G_k^k = a_k.b_k = (a_k \bar{\oplus} b_k).a_k = \bar{P_k^k}.a_k$, alors $G_k^0 = \bar{P_k^k}.a_k + P_k^k.G_{k-1}^0$ peut-être réalisé par un multiplexeur 2:1. La profondeur du Δarbre est **minimale** (1) mais la longueur de la Δtranche maximale (n) selon la figure 4.11. La retenue est en effet propagée de l'entrée a_0 jusqu'à la sortie $n-1$, passant à travers tous les modules $CY-MUX$. Le délai est la somme des *FMAP* initiaux et finaux, ainsi que la retenue de n *CY-MUX*.

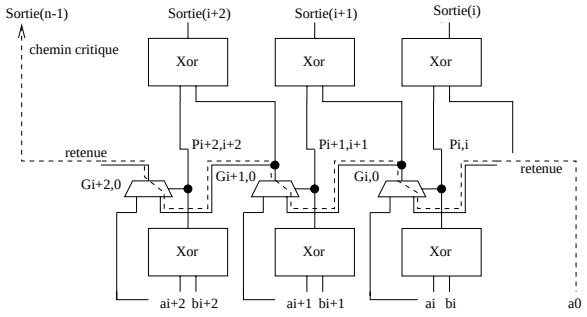

Figure 4.11: Additionneur série pour le FPGA Xilinx 5200.

L'implantation parallèle traduit l'additionneur de Sklanski en un réseau de module *FMAP*. Le Δarbre est **logarithmique** ($\lceil log_2(n) \rceil$) sans tenir compte des Xor initiaux

et finaux. Une réinjection sélective suivie d'une duplication permet de saturer les *FMAP* en en limitant la sortance. Les délais des deux architectures de base sont respectivement:

$$\begin{cases} t_{série} & = 2t_{FMAP} + n.t_{CY-MUX} \\ t_{parallèle} & \leq (\lceil log_2(n) \rceil + 2)t_{FMAP}. \end{cases} \tag{4.9}$$

Le délai de l'architecture parallèle est logarithmique tandis que celui de l'architecture série est linéaire. Mais si $t_{CY-MUX} << t_{FMAP}$, le parallelisme l'emportera pour les grandes tailles d'opérandes seulement. L'idée est de combiner de petites additions sérielles très rapides avec une profondeur logarithmique dans une solution hybride. Ceci est possible en construisant un Δarbre dichotomique à l'aide de Δtranches séries. Dans ce cas:

Propriété 4.5.1 (Profondeur du Δarbre) *Si toutes les Δtranches ont même longueur $\Delta longueur$, le Δarbre a une profondeur de $\lceil log_{\Delta longueur}(n) \rceil$.*

Les termes P_i^j et G_i^j doivent être construis avec des modules *CY-MUX*. Les équations du premier niveau sont:

$$\begin{cases} P_i^j & = P_i^i.P_{i-1}^j & = P_i^i.P_{i-1}^j + \bar{P_i^i}.0 \\ G_i^j & = P_i^i.G_{i-1}^j + G_i^i & = P_i^i.G_{i-1}^j + \bar{P_i^i}.a_i. \end{cases} \tag{4.10}$$

et celles des autres tranches:

$$\begin{cases} P_i^j & = P_i^k.P_{k-1}^j & = P_i^k.P_{k-1}^j + \bar{P_i^k}.0 \\ G_i^j & = P_i^k.G_{k-1}^j + G_i^k & = P_i^k.G_{i-1}^j + \bar{P_i^k}G_i^k. \end{cases} \tag{4.11}$$

(car $P_i^k.G_i^k = 0$.) Par conséquent, toutes les équations peuvent-être exprimées à l'aide de multiplexeurs 2:1. Un schéma de principe est fourni en figure 4.12: pour une addition de neuf bits. les Δtranches $\Delta[8,6]$, $\Delta[5,3]$ et $\Delta[2,0]$ sont calculées en parallèle. Ensuite, ces petits additionneurs sont utilisés pour former la tranche $\Delta[8,0]$, également obtenue par un additionneur de 3 bits. Remarquons que puisque la sortie d'un $CY-MUX$ ne peut être utilisée que par l'entrée du $CY-MUX$ suivant, il est nécessaire de dupliquer cet additionneur plusieurs fois. Par exemple, la sortie 3, obtenue par l'additionneur $G_{3,0}$ ne peut être obtenue par l'additionneur $G_{4,0}$. La retenue traverse alors deux étages d'additionneurs de longueur 3. Notons toutefois que cette solution ne sera choisie que pour des opérandes de plus grandes tailles.

Le délai de l'additionneur hybride est estimé comme suit: le chemin critique passe par une Δtranche série pour chacun des $\lceil log_{\Delta longueur} \rceil$ niveaux. Toutes les Δtranches ont une longueur Δlongueur exceptée la dernière qui peut-être insaturée, soit un délai de $\Delta longueur.t_{CY-MUX} + t_{FMAP}$ pour les premières $\lceil log_{\Delta longueur} \rceil - 1$ tranches. La dernière tranche ne comporte en effet que $\lceil \frac{n}{\Delta longueur^{\lceil log_{\Delta longueur} \rceil - 1}} \rceil$ entrées. Le délai total est donc:

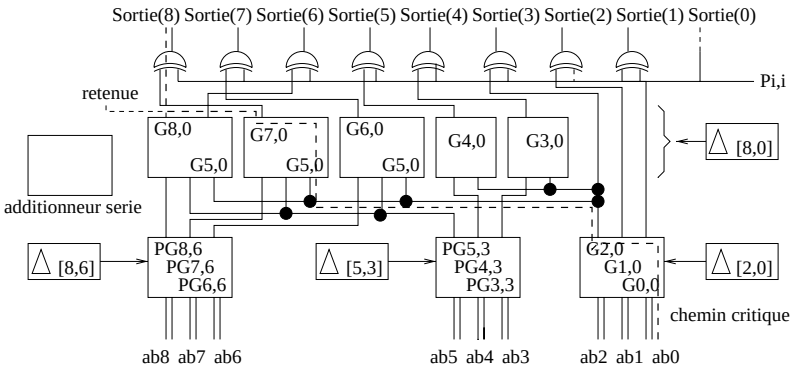

Figure 4.12: Additionneur hybride de 9 bits réalisé avec des Δtranches de 3 bits.

$$\begin{cases} t_{hybride} & = t_{FMAP} + (\lceil log_{\Delta longueur} \rceil - 1)(\Delta longueur.t_{CY-MUX} + t_{FMAP}) + t_{dernière}, \\ t_{dernière} & = \lceil \frac{n}{\Delta longueur^{\lceil log_{\Delta longueur} \rceil - 1}} \rceil .t_{CY-MUX} + t_{FMAP}. \end{cases} \tag{4.12}$$

La Δlongueur **optimale** minimisant (4.12) est calculée empiriquement. Lorsque $n \leq 17$, la solution hybride est l'additionneur série. Mais lorsque $n > 17$, la profondeur logarithmique est meilleure. Des valeurs typiques de t_{FMAP} et t_{CY-MUX} sont issues des expériences précédentes pour prendre en compte les délais d'interconnexions, qui peuvent représenter deux à trois fois le délai logique dans un tel FPGA. Une étude plus complète se trouve dans [43]. Les trois solutions ont été implantées dans notre outil de synthèse et placées et routées. Le *part type* utilisé est $5215PG299$ avec le *speed grade 3*. Notons que les résultats contiennent les buffers d'entrée/sortie si bien que le délai réel de l'addition est plus faible. La solution hybride est meilleure de 20% par rapport à l'additionneur parallèle pour 80 bits. Pour les opérandes de grandes tailles, la solution hybride est 2.25 fois plus rapide que l'additionneur série. La surface des additionneurs parallèles et hybrides sont deux et trois fois plus gros que l'additionneur série respectivement. La raison est qu'un module *CY-MUX* ne peut communiquer qu'avec le module *CY-MUX* suivant, si bien que beaucoup d'entre eux doivent être dupliqués lorsque leurs sorties sont utilisées plusieurs fois. (Voir le *CY-MUX* G_0^5 de la tranche $\Delta(8,0)$ dans la figure 4.12.) Les résultats des délais de placement et routage sont fournis à la fin de chapitre.

4.5.3 Optimisation pour le CPLD AMD Mach 5

Les CPLD ne permettent que la formation de Δtranches parallèles puisqu'il n'y a pas de particularités spécifiques quant à la propagation sérielle des retenues. Par contre, on peut construire la plus grande Δtranche parallèle possible en utilisant un seul niveau de macro-cellule. Pour ce faire, les équations des additionneurs *carry look-*

ahead peuvent être réinjectées pour obtenir les termes de propagation et de génération en une somme de monômes:

$$\begin{cases} P_i^j &= P_j^j.P_{j+1}^{j+1}.P_{j+2}^{j+2}.\cdots.P_i^i, \\ G_i^j &= G_i^i + G_{i-1}^{i-1}.P_i^i + G_{i-2}^{i-2}.P_{i-1}^{i-1}.P_i^i + \cdots + G_j^j.P_{j+1}^{j+1}.P_{j+2}^{j+2}\cdots P_i^i. \end{cases} \tag{4.13}$$

Les limitations du nombre de variables et de monômes ne permettent pas de contruire des Δtranches de plus de 16 bits (équation (4.13)). L'étage de *Xor* final peut-être immergé dans la dernière macro-cellule puisque celle-ci permet de réaliser un *Ou* exclusif avec l'une de ses entrées. Par contre, l'étage de *Xor* initial doit être inclu dans le premier niveau de macro-cellules. Malheureusement, les produits de *Xor* génèrent un nombre exponentiel de monômes, si bien qu'une tranche d'au plus quatre bits peut être construite en un niveau de macro-cellules. (le terme de propagation génère en effet $2^4 = 16$ monômes.) La solution résultante est la plus rapide possible en terme de profondeur de macro-cellules. Notons toutefois qu'une heuristique hybride (sérialisation de Δtranches parallèles) sera le plus souvent utilisée dans l'outil ASYL+ pour réduire l'accroissement important de la surface. La profondeur de macro-cellule donnée ci-dessous permet, par exemple, de réaliser un additionneur de 64 bits en deux niveaux de macro-cellules.

$$t_{parallèle} = \lceil \frac{\lceil \frac{n}{4} \rceil}{16} \rceil + 1. \tag{4.14}$$

4.6 Les multiplieurs classiques

4.6.1 Algorithmes de multiplication

La multiplication est impliquée dans 10% environ de toutes les instructions d'un programme scientifique typique [44]. Elle exige souvent plusieurs cycles pour s'effectuer et peut être séquentielle. Concevoir des multiplieurs rapides est donc crucial pour l'efficacité des processeurs. Ils consistent à accumuler des produits partiels [45]. Par conséquent, deux étapes majeures doivent être optimisées: la **génération** des produits partiels et leur **accumulation**. Accumuler deux niveaux de produits partiels est une addition classique si bien que celle-ci est la troisième étape de l'algorithme. Un seul type de multiplieur est étudié ici, qui est le plus efficace: le multiplieur parallèle, qui génère en parallèle les produits partiels et les accumule par un additionneur à opérandes multiples. Trois méthodes sont alors utilisées pour accélérer la multiplication: (1) produire moins de produits partiels, (2) les accumuler le plus rapidement possible et (3) accélérer la propagation de la retenue dans l'additionneur final. Nous proposons donc par la suite de revoir en détail chacune de ces étapes et de comparer les architectures classiques d'un point de vue de la vitesse et de la puissance consommée.

4.6.2 Encodage des opérandes

L'algorithme *add/shift* ajoute conditionnellement le multiplicande suivant la valeur du bit du multiplieur. La logique de sélection, une porte *Et*, a l'avantage d'être l'approche la plus rapide mais produit le plus grand nombre de produits partiels.

L'encodage (modifié) de Booth [46][47] partitionne le multiplieur en groupes de trois bits recouvrants. Chaque groupe est décodé en parallèle pour sélectionner un multiple du multiplicande M dans l'ensemble $\{0, \pm\mathrm{M}, \pm 2\mathrm{M}\}$. Heureusement, tous ces multiples sont atteints par simple décalage et complémentation. La complémentation à deux nécessite en général d'ajouter une série de 1 si le multiple est négatif. Mais elle peut s'effectuer plus simplement comme indiqué sur la figure 4.13). Les points noirs représentent les bits à ajouter. Chaque groupe de 3 bits du multiplieur permet de générer un opérande. La complémentation à deux est en effet réalisée en rajoutant astucieusement des bits de signe s, sauf pour le dernier. En effet, l'ajout de 0 assure que le dernier multiple soit positif. Le nombre d'opérandes à sommer est réduit à $\lceil \frac{n+1}{2} \rceil$ puisque deux nouveux bits sont examinés pour chaque groupes de trois bits recouvrants.

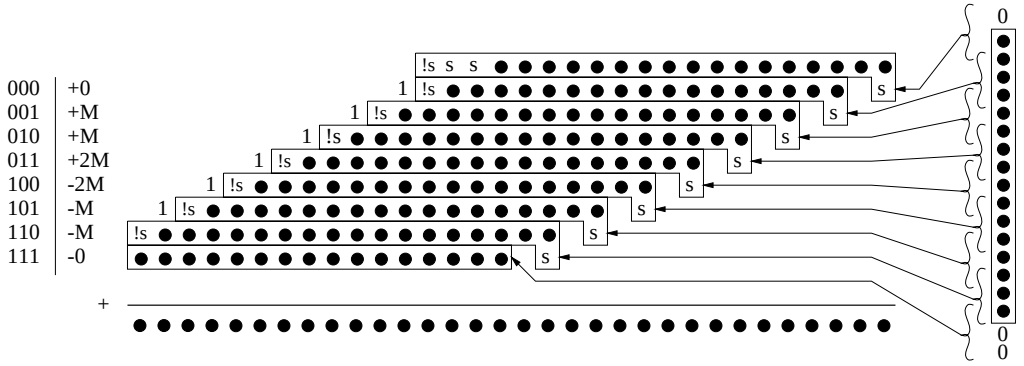

Figure 4.13: Encodage de Booth radix 2.

Lorsque plus de bits sont examinés en même temps, par exemple quatre, une meilleure réduction est possible mais au prix de la génération des multiples $\{0, \pm\mathrm{M}, \pm 2\mathrm{M}, \pm 3\mathrm{M}, \pm 4\mathrm{M}\}$ (*Booth radix 3*). Le multiple $3M$ n'est malheuresement atteint que par l'addition de $2M$ et M. Le nombre d'opérandes n'est que de $\lceil \frac{n+1}{3} \rceil$ mais requiert une propagation de retenue sur une longueur de n. (Dans le cas général de l'algorithme de *Booth radix p*, le nombre d'opérandes est $\lceil \frac{n+1}{p} \rceil$.) Pour obtenir un compromis entre la réduction et la vitesse d'encodage, Bewick et Flynn proposent dans [48] de fractionner l'addition $3M$ en petits additionneurs. Cette solution à redondance partielle exige quelques opérandes supplémentaires remplies par les retenues des petits additionneurs qui recouvrent les sorties des additionneurs suivants. Au lieu de complémenter la somme $\pm(2M + M)$, nous avons ajouté les multiples déjà

complémentés $\pm 2M + \pm M$ de telle sorte qu'un opérande supplémentaire est supprimée par rapport à l'approche originale. La taille des additionneurs est choisie pour que les retenues intermédiaires des additonneurs ne se chevauchent pas: une taille de quatre est la plus petite longueur adéquate. Une solution intermédiaire entre *Booth 2* et *Booth 3 à redondance parielle* peut être réalisée en produisant la moitié des opérandes selon chacun des algorithmes.

Finalement, nous pouvons aussi fractionner les deux opérandes en groupes de p bits et les multiplier deux à deux. Le décalage de ces petites multiplications produit alors un jeu d'opérandes équivalentes. Pour un encodage compétitif par rapport aux algorithmes précédents, une architecture traditionnelle de *Braun* orientée surface est utilisée, qui fournit néanmoins une bonne réduction avec $\lceil \frac{n}{4} \rceil - 1$ opérandes.

4.6.3 Réduction des produits partiels

Pour réduire le nombre d'opérandes à deux, une addition *carry save* est appliquée. Pour toutes les topologies, le même nombre de compteurs $(3, 2)$ (*full adder*) est utilisé mais beaucoup d'interconnexions sont réalisables. Les deux grandes familles se distinguent par la sérialisation ou la parallélisation de l'arbre d'addition.

Les arbres séries sont des combinaisons d'additions **linéaires** (*linear arrays*) dans lesquels les produits partiels sont additionnés séquentiellement à ceux déjà accumulés. Les *double linear arrays* [49] additionnent produits partiels pairs et impairs en parallèle tel que le délai de réduction est diminué de moitié. Zuras et McAllister propose de combiner des portions linéaires de longueurs croissantes: une portion est connectée au reste de l'arbre *ZM* lorsque leurs délais sont identiques [50]. Encore plus efficace et souvent optimal, l'arbre *Overturned Stairs (OS)* de Mou et Joutand [51] est récursivement construit avec des arbres *OS* plus petits et des portions linéaires.

D'un autre côté, on peut éliminer les contraintes temporelles des accumulateurs linéaires en additionnant tous les produits partiels en **parallèle**: Dadda propose dans [52] de construire des arbres de Wallace optimaux à l'aide de compteurs $(3, 2)$ et $(2, 2)$. Le nombre de niveaux de compteurs suit les éléments de la série:

$$\begin{cases} U_0 & = 2, \\ U_{n+1} & \lfloor \frac{3U_n}{2} \rfloor. \end{cases} \tag{4.15}$$

On peut également former un compteur $(4, 2)$ à l'aide de deux compteurs $(3, 2)$ et construire un arbre logarithmique avec un niveau de ces compteurs pour chaque puissance de deux d'opérandes. Il peut être un peu moins performant que la réduction de Dadda puisqu'une contrainte supplémentaire est imposée sur les interconnexions de compteurs $(3, 2)$. Notons que puisque toutes les topologies précédentes ont la même sortance, le délai des arbres de réduction dépend uniquement du nombre de compteurs

contenus sur le plus long chemin. Nous fournissons en figure 4.14 la représentation des arbres de réductions précédents pour quatorze opérandes à additionner. Elles exigent à peu près le même nombre de niveau de compteurs, mais ces architectures doivent rapidement se distinguer pour des opérandes de plus grandes tailles.

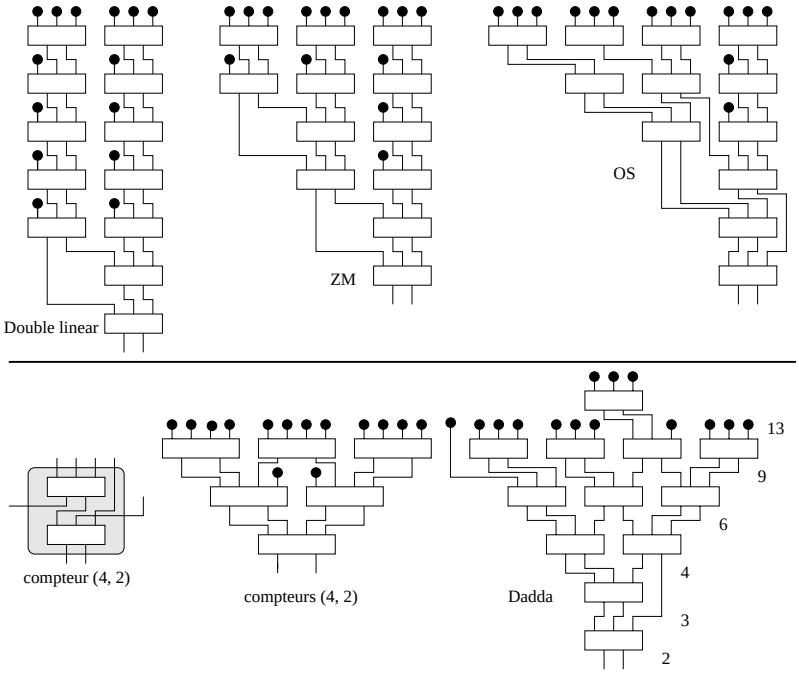

Figure 4.14: Topologies de réduction.

4.6.4 Résultats expérimentaux

Les résultats expérimentaux ont été obtenus avec la bibliothèque sub-micronique $0.8\mu m$ de *Thomson TCS*. La plus petite latence d'encodage est obtenue par l'algorithme de *Add/Shift*. L'encodage de Booth est rapide puisqu'il ne requiert pas la génération de multiples complexes avec une logique de sélection assez simple. L'algorithme de Booth *radix 3* est à éliminer du à l'accroissement linéaire de la latence. L'addition à redondance partielle est un succès puisqu'il est meilleur que la multiplication décomposée, par le nombre d'opérandes ainsi que par la vitesse d'encodage. La solution hybride *Booth 2/3* n'amène rien de plus. Pour le critère de puissance, toutes les solutions à base d'additions *ripple carry* doublent ou triplent la puissance dissipée. Pour la phase de réduction, la différence entre structures linéaires et parallèles est évidente: les architectures rapides sont aussi celles qui consomment le moins. L'explication est fort simple: l'opérateur *Xor*, constituant principal des compteurs, propage très bien les transitions inutiles puisque $a \oplus b \neq \bar{a} \oplus b$. Comme les compteurs des architectures linéaires ont tous des temps d'arrivées différents, il peut y avoir autant de

transitions qu'il y a de niveaux de compteurs précédents. Par conséquent, plus l'arbre de réduction est linéaire, plus il dissipe de puissance (jusqu'à neuf fois plus de puissance inutile que de puissance logique pour 32 opérandes !). Les critères de surface et de comsommation sont donc distincts puisque les meilleurs multiplieurs surface (i.e. multiplieur de Braun à base d'addition *carry save*) sont ceux qui consomment le plus. Les structures parallèles ont entre 20% et 25% de glitches, le minimum étant obtenu pour la structure très régulière à base de compteurs $(4, 2)$. Les probabilités de commutations ont été obtenues par l'agorithme *Add/Shift* mais les autres encodages peuvent produire plusieurs transitions donc augmenter la consommation. La comparaison finale en tiendra compte. L'addition finale montre aussi fort bien la différence entre propagation sérielle et dichotomique. Seules les architectures *carry look-ahead* sont viables malgré une augmentation inquiétante (exponentielle) de la sortance de l'architecture de Sklanski qui nécessite donc des cellules bufferisées. La dissipation de puissance est fonction de la complexité de l'architecture et on retrouve le rapport entre puissance inutile et totale de l'ordre de 30% pour l'additionneur *ripple carry*.

Les algorithmes les plus efficaces (4 encodages, 2 réductions et 3 additionneurs) donnent 24 solutions que nous avons synthétisées jusqu'à 64 bits et illustrées en figure 4.15: Les paramètres utilisés sont le délai en ns et la puissance consommée, calculée comme le produit de la probabilité de commutation et la capacité attaquée. On y a représenté, par des figures géométriques, les algorithmes d'encodage (décalage, encodage de Booth de radix 2 et 3, et utilisation de petits multiplieurs), par des couleurs les arbres de réductions utilisés (arbres de Wallace et réduction de Dadda). Le lecteur est aussi amené à déduire, de cette représentation, la position des solutions obtenues en remplaçant l'additionneur de Han-Carlson par les architectures de Sklanski et Kogge-Stone. Il faut alors rajouter entre 3 et 8 ns pour le premier, et 20 à 200 *pc* pour le second. Les meilleures architectures consomment entre 33% et 50% de transitions inutiles: le meilleur compromis latence/consommation est obtenu par l'agorithme *Add/Shift* puisque sa vitesse est la meilleure (égale au *Booth 2* pour les grandes opérandes) alors que la puissance consommée est moindre avec 33% de transitions inutiles. Remarquons que le placement et routage des multiplieurs avantage les encodages de Booth puisque la longueur des connexions reliant les multiplexeurs d'encodage et les premiers compteurs est plus petite car le nombre de compteurs est aussi plus petit. Les solutions de Booth à fort radix sont médiocres puisque les additions sérielles équivalent à quatre niveaux de compteurs. Les arbres de Wallace sont meilleurs de 10% par rapport aux compteurs $(4, 2)$. Pour l'addition finale, l'architecture de Han et Carlson représente le meilleur compromis avec une sortance optimale et une complexité réduite par rapport aux architectures de Sklanski et Kogge et Stone respectivement. Le schéma 4.15 résume mieux qu'un tableau de chiffres les comparaisons effectuées. Il est aisé de voir que les différences d'encodage, de réduction et de sortance prennent une importance considérable avec la taille croissante des

opérandes. De ces résultats expérimentaux, nous en déduisons que la meilleure combinaison pour le compromis vitesse/consommation est l'encodage *Add/Shift* suivi d'un arbre de Dadda et d'une addition Han et Carlson.

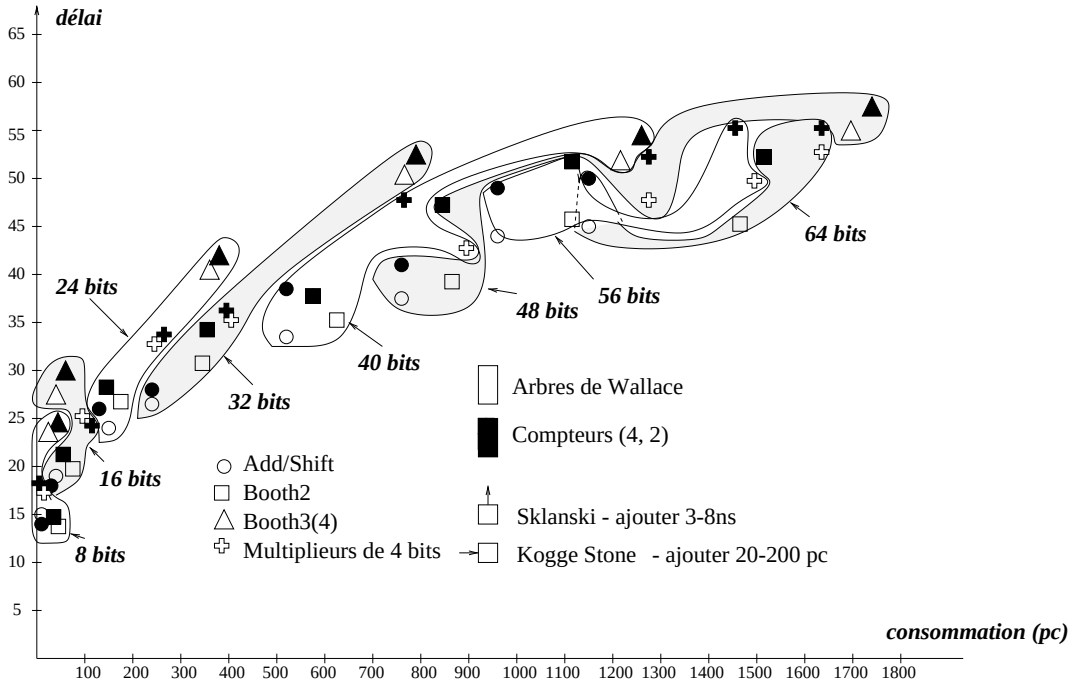

Figure 4.15: Comparaison des multiplications complètes.

4.7 Optimisation de la multiplication sous contraintes technologiques

Les mêmes technologies que pour l'addition sont utilisées pour la multiplication. Pour les CPLD, les deux encodages de Booth et de décomposition de la multiplication sont efficacement implantés, suivis des arbres de Wallace et de l'additionneur rapide des sections précédentes. Puisque le niveau de macro-cellule doit être minimisé, nous sommes arrivés à: (1) implanter l'encodage de Booth sur un seul niveau de macro-cellule, (2) implanter un multiplieur de Braun [53] de trois bits sur un niveau de macro-cellule (on en utilise en tout $\lceil \frac{n}{3} \rceil^2$), (3) implanter un niveau de réduction de Dadda sur un niveau de macro-cellule. Le nombre d'opérandes produits par les encodages sont respectivement de $\lfloor \frac{n}{2} + 1 \rfloor$ et $2\lceil \frac{n}{3} \rceil - 1$. Par conséquent, le délai de ces multiplieurs est exprimé ci-dessous. C'est pourquoi la solution de Booth ne sera retenue que pour les multiplications de plus de 8 bits.

$$\begin{cases} t_{Booth} & = 2 + Dadda(\lfloor \frac{n}{2} + 1 \rfloor) + \lceil \frac{\lceil \frac{2n}{4} \rceil}{16} \rceil, \\ t_{décomposition} & = 2 + Dadda(2\lceil \frac{n}{3} \rceil - 1) + \lceil \frac{\lceil \frac{2n}{4} \rceil}{16} \rceil. \end{cases} \quad (4.16)$$

Les additionneurs sur *FPGA Xilinx 5200* ont mis en évidence l'utilité de l'approche technologique sérielle. La multiplication peut donc aussi tirer profit de cette approche structurelle en décomposant la multiplication en un arbre dichotomique d'additionneurs binaires [43b], comme illustré sur la figure 4.16: les additionneurs sont de tailles croissantes puisque les retenues générées à un étage font partie de l'étage suivant. Les additionneurs séries sont conçus en utilisant deux techniques:

- Un étage est partiellement réutilisé dans le suivant. Le terme de propagation s'appuie sur les sorties de l'étage précédent (selon $S_k \oplus S'_k$) qui peuvent être réinjectées en $(G^0_{k-1} \oplus P^k_k) \oplus (G'^0_{k-1} \oplus P'^k_k)$ et contenues dans un LUT à quatre entrées. Les sorties S_k et S'_k sont tout de même utilisées, mais au niveau de $FMAP$ suivant.

- Les retenues ne se propagent pas à travers tous les additionneurs d'une branche de l'arbre. Puisque celles-ci se propagent de la même façon pour tous les étages, une seule d'entre elles est comptabilisée dans le chemin critique.

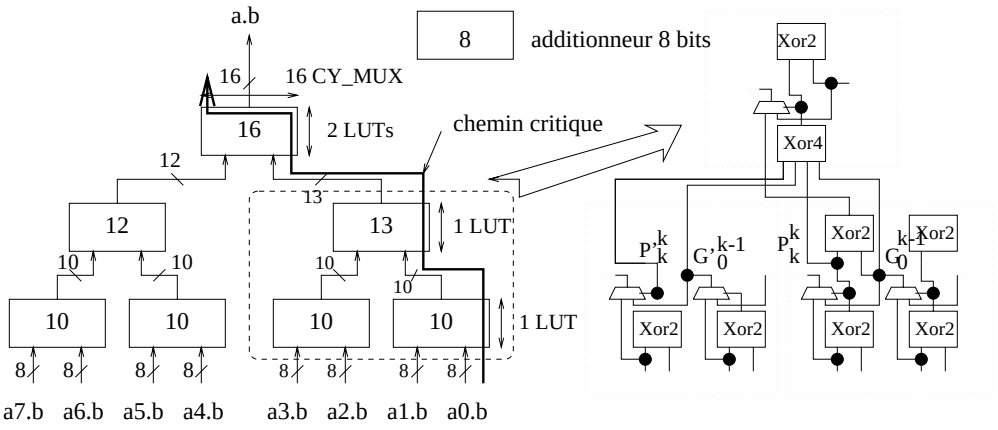

Figure 4.16: Multiplieur structurel de 8 bits.

Le délai théorique est donné ci-dessous. Notons toutefois que puisque chaque étage produit $2n$ connexions avec les étages suivants, cette architecture peut souffrir de problèmes de routage.

$$t_{structurel} = [log_2(n) - 1].t_{FMAP} + t_{série}. \quad (4.17)$$

De même que pour les CPLD, l'algorithme de Booth/Wallace est implanté en utilisant deux *FMAP* pour l'encodage et une pour une étape de réduction respectivement. En réécrivant la série géométrique de Dadda en $U_N \approx 2(\frac{3}{2})^N$, le délai de cette architecture classique est de:

$$t_{Booth} \approx (1 + \frac{ln\frac{n}{4}}{ln\frac{3}{2}}).t_{FMAP} + t_{parallèle}. \tag{4.18}$$

Pour les opérandes de tailles modérées, les additionneurs sérielles et parallèle ont le même délai donc la vitesse des deux architectures dépend du terme $log_2(n) - 2 - \frac{ln\frac{n}{4}}{ln\frac{3}{2}}$ qui est négatif pour $n \geq 5$. L'approche structurelle semble donc la plus performante. Malheureusement, les pénalités de connectivité rendent cette architecture non routable pour plus de 18 bits (le multiplieurs 13 bits est même incomplètement routé). Mais puisque cette approche est deux fois plus petite que son équivalente algorithmique, elle sera choisie chaque fois que la taille de l'opérande est faible. Autrement, le multiplieur de Booth/Wallace sera retenu. La figure suivante résume graphiquement les résultats expérimentaux obtenus à l'aide des outils de placement et routage de la société *Xilinx*. Elle montre clairement le comportement linéaire ou logarithmique des différents additionneurs ainsi que la comparaison entre l'approche structurelle et algorithmique des multiplieurs.

Nous fournissons donc en fin de chapitre l'ensemble des expérimentations effectuées sur la cible Xilinx 5200. On a représenté les performances des additionneurs et des multiplieurs. L'objectif est de bien visualiser le délai linéaire de la solution série existante et les délais logarithmiques des deux solutions proposées. La différence est trèse importante pour 80 bits.

4.8 Conclusion

Nous avons présenté la structure des macro-blocs classiques utilisés dans un macro-générateur de synthèse RTL. Le formalisme de l'addition que nous avons introduit permet à l'outil de satisfaire au mieux les contraintes temporelles ou technologiques. Les multiplieurs peuvent également trouver un bon compromis entre les contraintes technologiques, les contraintes temporelles et de consommation.

Chapitre 5

Conception de Blocs de Codage Correcteur d'Erreur

5.1 Introduction

La synthèse comportementale découpe les pas de calcul en ensembles de cycles d'horloge pour fournir une architecture synchrone. Une fois le code source débarassé de tous les détails relatifs au langage de description, la synthèse comportementale réalise l'ordonnancement, l'allocation, puis l'allocation des connexions.

Plus le niveau de description est élevé, plus le concepteur a le désir d'exprimer, via la bibliothèque de blocs, des opérations complexes. Ainsi, il pourra en un minimum d'effort concevoir un circuit de grande taille. L'utilisation des blocs réutilisables dans la phase de synthèse comportementale a été développée au chapitre 2. La seconde partie du travail consiste donc à construire une bibliothèque de blocs réutilisables. C'est justement la contribution de ce chapitre. La figure 5.1 illustre l'inférence d'un bloc de codage remplaçant un pas de calcul dans une description comportementale. Le code RTL produit intègre donc la description correspondante.

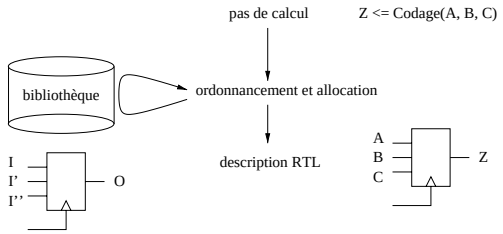

Figure 5.1: L'étape de réutilisation en synthèse comportementale.

Nous avons conçus un ensemble de blocs comportementaux utilisables dans une

même application. Ils s'articulent autour de la redondance présente dans l'information transmise par les systèmes de communication. On l'enlève afin de diminuer la taille de l'information à transmettre, mais on en ajoute pour assurer l'intégrité des données. Il s'agit des opérations de correction d'erreur et de compression. L'objectif de ce chapitre est triple:

- Construire tous les blocs présents dans une application spécifique développée à TCS, à savoir un système de communication d'images muni d'un codage correcteur d'erreur complexe.
- Proposer à des clients éventuels un éventail homogène d'IP's pouvant satisfaire toutes les contraintes de surface ou de débit.
- Valider notre méthodologie de conception sur des applications industrielles.

Le chapitre est composé comme suit: suite à une brêve introduction de la theorie du codage, le chapitre présente les codes convolutionnels puis les codes de Reed-solomon. Pour chacun d'entre eux, on présente les architectures implantées, la méthodologie de conception, puis les résultats expérimentaux obtenus.

5.2 Théorie du codage

Nous présentons ici très brièvement la théorie du codage. Le lecteur est encouragé à lire un des ouvrages suivants pour trouver toutes les bases mathématiques des codes correcteurs d'erreurs [54][55][56].

5.2.1 Transmission avec bruit

La théorie du codage a pour but d'assurer l'intégrité des données lors du transfert d'information. Le medium physique à travers lequel les données sont transmises s'appelle le canal. Malheureusement, le bruit dans le canal altère l'information et la théorie du codage doit détecter et corriger de telles corruptions. Le diagramme suivant présente un système de transmission d'information. Sur la figure 5.2, l'information, composée d'une chaine de bits, peut être corrompue le long du canal de transmission. Le décodeur localise alors le bit erroné et retrouve sa valeur initiale.

L'ensemble de tous les mots valides sont les mots du code. La détection et la correction d'erreurs éventuelles n'est possible que si les mots du code ne couvrent pas tous les mots possibles sans quoi tous les mots reçus en sortie de canal sont valides et sont donc considérés avoir été transmis sans erreur. Les bits dans le canal ont une probabilité d'être correctement transmis plus grande que d'être altérés, si bien que le mot du code transmis le plus probable est celui qui diffère du mot reçu du moins de positions possibles.

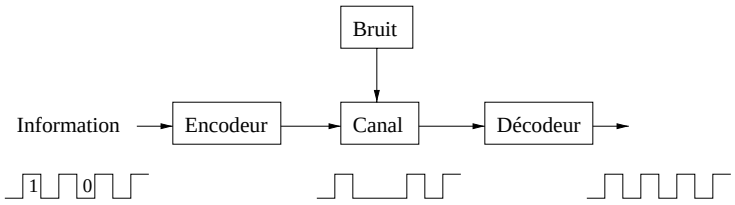

Figure 5.2: Système de transmission d'information.

La distance d'un code $\mathcal{C}$ est le plus petit nombre de positions dont peuvent différer deux mots quelconques du code. Les cercles suivants (ou sphères de décodage) représentent tous les mots qui sont à une distance de $\frac{d-1}{2}$ au plus d'un mot du code. Alors toutes les erreurs de poids $d-1$ au plus peuvent être détectées parceque le mot corrompu ne peut atteindre un autre mot du code, et toutes les erreurs de poids $\frac{d-1}{2}$ au plus peuvent être corrigées puisqu'il existe un seul mot du code le plus proche possible du mot reçu. La figure 5.3 montre trois positions possibles d'un mot corrompu dont la valeur initiale est le centre de la sphère hachurée. Le mot corrompu peut être à l'intérieur de la sphère, donc il sera correctement corrigé. Il peut être contenu dans une autre sphère, donc il sera corrigé en un mot qui n'était pas le message initial. Ou bien il peut se trouver entre deux sphères. Dans ce cas, l'erreur sera détectée mais ne pourra être corrigée.

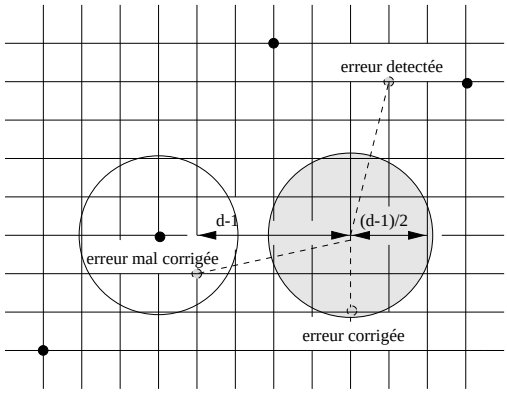

Figure 5.3: Distance d'un code.

Pour créer une distance minimale entre les mots du code, plus de bits sont utilisés que ceux nécessaires à coder l'information. Les bits additionnels sont appelés **redondance**. On essaie de concaténer la redondance à l'information pour la rendre visible et directement disponible. La qualité d'un code se mesure donc par la distance qu'il procure par rapport au nombre de symboles de redondance qu'il nécessite.

5.2.2 Codes correcteurs d'erreur

La structure algébrique des codes fournit un environnement dans lequel des codes et des algorithmes de codage peuvent être efficacement construits. Lorsque le code $\mathcal{C}$ est un sous-espace vectoriel, c'est-à-dire si $v \in \mathcal{C}$ et $w \in \mathcal{C}$ alors $v + w \in \mathcal{C}$, le code est dit **linéaire**. A la fois $\mathcal{C}$ et $\mathcal{C}^{\perp}$ (espace dual de $\mathcal{C}$) peuvent être décrit à l'aide de matrices de vecteurs d'une base, si bien qu'encodage et décodage sont effectués par des produits vectoriels et matriciels. Lorsque $\mathcal{C}$ est un sous-espace cyclique, c'est-à-dire si $(a_1, a_2, \cdots, a_{n-1}, a_n) \in \mathcal{C}$ alors $(a_n, a_1, \cdots, a_{n-2}, a_{n-1}) \in \mathcal{C}$, le code est dit **cyclique**. L'avantage est de pouvoir représenter l'information sous forme de polynômes et d'effectuer encodage et décodage par des produits et des divisions polynômiales. La représentation et les algorithmes de codage sont ainsi plus concis.

Les codes de Reed-Solomon utilisent les éléments d'un corps de Galois pour transporter l'information au lieu de bits. Un corps de Galois est un groupe abélien pour l'addition et la multiplication (privée de l'élément nul) avec une propriété distributive. Les entiers modulo un nombre premier n (Z_n) est un corps de Galois. Les classes de congruence des polynômes modulo un polynôme irréductible sur Z_n est aussi un corps de Galois. Dans chaque corps il existe un élément primitif dont les puissances successives forment le corps tout entier et pour chaque élément non nul α d'un corps de Galois fini $GF(q)$ composé de q éléments, $\alpha^{q-1} = 1$. Par exemple, le corps de Galois $GF(2^3)$ modulo le polynôme $f(x) = 1 + x + x^3$ est composé des éléments 000, 001(α^0), 010(α^1), 100(α^2), 011(α^3), 110(α^4), 111(α^5), 101(α^6), écrit comme les coefficients d'un polynôme $b_2x^2 + b_1x + b_0$.

Soit β un élément d'un corps de Galois $GF(p^q)$ et une racine primitive $n^{ième}$ de l'unité.

$$g(x) = (x - \beta^{1+a})(x - \beta^{2+a}) \cdots (x - \beta^{\delta-1+a}) \tag{5.1}$$

(avec $\delta \geq 2$ et $a \geq 0$) génère un code linéaire cyclique appelé code de Reed-Solomon sur $GF(p^q)$ de distance δ. Puisque le nombre de symboles de redondance est $n - k = \delta - 1$, ce code atteint la **limite** théorique **de Singleton**, c'est-à-dire qu'il n'existe pas de code corrigeant plus d'erreurs avec le même nombre de symboles de redondance. Cette propriété en fait un code très puissant: il peut corriger $\frac{\delta-1}{2}$ erreurs, $\delta - 1$ effacements (erreur dont on connait la position). De plus, les codes de Reed-Solomon sont très efficaces pour les corrections de rafales d'erreurs puisq'un code corrigeant t erreurs peut aussi corriger une rafale de $q(t-1)+1$ bits.

Alors que les codes de Reed-Solomon corrigent très bien les rafales d'erreurs, ils sont mis en échec lorsque les erreurs sont éparpillées. Par conséquent, ils sont souvent entrelacés à l'aide d'un code convolutionnel. Un code convolutionnel sur un corps fini F est obtenu en convoluant la séquence d'entrée par un circuit constant, linéaire, causal et fini. La fonction de transfert $T(D)$ de l'encodeur a des composants qui sont

des fonctions réalisables, c'est-à-dire le rapport de polynômes $\frac{h}{\phi}$ avec $deg(h) \leq deg(\phi)$. Tout encodeur est équivalent à un encodeur basique (sans rétroaction dans l'encodeur ou dans son inverse) et minimal (contenant le nombre minimal de points mémoires) [57]. Pour un code convolutionnel à une séquence d'entrée et n séquences de sortie $(1, n)$, une condition nécessaire et suffisante pour que l'encodeur soit basique et minimal est que le plus grand commun diviseur (pgcd) des polynômes générateurs h soit égal à 1. Le degré du plus grand polynôme est la contrainte du code.

5.3 Codes convolutionnels

5.3.1 Structure de code

Un encodeur $(1, n)$ de contrainte v a 2^{v+1} états physiques. En associant le symbole d'entrée à la transition, un diagramme d'état de 2^v états et de 2^{v+1} transitions peut-être construit. Puisque la structure devient répétitive après $v+1$ branches, le diagramme peut-être redessiné en treillis [58], qui peut-être lui-même réduit. La figure 5.4 montre plusieurs représentations du même code convolutionnel. La représentation de gauche est celle d'un encodeur convolutionnel où est indiquée la fonction logique réalisée sur le registre à décalage, afin de calculer les deux synonymes (sorties). Au centre, on représente le contenu de ces v registres par un code de $v + 1$ bits. L'entrée d'un nouveau bit dans le registre à décalage entraine deux transitions possibles entre les états, et un treillis est alors formé. Puisque celui-ci est périodique, la figure de droite n'en représente que les états et les transitions.

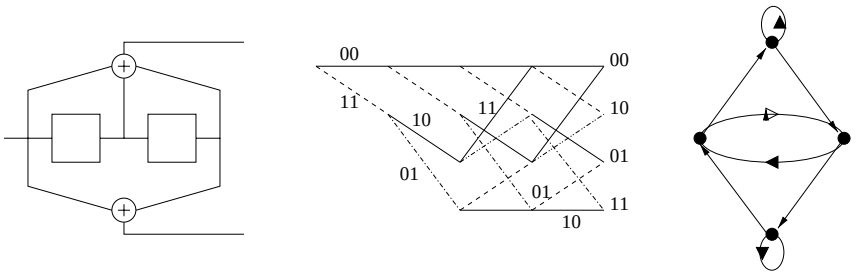

Figure 5.4: Représentations d'un code $(1, 2)$.

5.3.2 Décodage de Viterbi

Dans une machine de Markov, l'algorithme de décodage cherche à trouver la séquence d'états y, observant la séquence z pour laquelle la probabilité $P(y, z) = P(y|z)P(z)$ est maximisée:

$$-lnP(y,z) = \sum -lnP(y_{k+1}|y_k) - lnP(z_k|y_{k+1},y_k) \tag{5.2}$$

puisqu'un nœud du treillis a plusieurs chemins convergents. Le chemin pour lequel la métrique est maximisée est le survivant [59] alors que les autres sont éliminés. On mémorise donc 2^v survivants et leur métrique au temps k. A $k+1$, on étend tous les chemins d'un état et on sélectionne les meilleurs sans jamais en mémoriser plus de 2^v. Ce choix de chemin peut également être modélisé par la solution de programmation dynamique de tous les chemins possibles [60]. Dans l'algorithme suivant, la métrique peut-être remplacée par la distance de Hamming.

Algorithme 2 - Décodage de Viterbi

1. Stockage: $\begin{cases} \textit{index de temps} & k \\ \textit{survivant en } s_k & y(s_k), \\ \textit{métrique de } y(s_k) & \Lambda(s_k) \end{cases}$

2. Initialisation: $\begin{cases} k = 0 \\ y(s_0) = s_0 \quad y(m) \textit{ arbitraire pour } m \neq s_0 \\ \Lambda(s_0) = 0 \quad \Lambda(m) = \infty \end{cases}$

3. Récursion: $\begin{cases} \Lambda(s_{k+1}, s_k) = \Lambda(s_k) + \lambda(y_{k+1}, z_{k+1}) \\ \Lambda(s_{k+1}) = min\ \Lambda(s_{k+1}, s_k)\ et\ stocker\ \ y(s_{k+1}). \end{cases}$

5.4 Organisation du décodeur

5.4.1 La récursion *Add-Compare-Select(ACS)*

L'algorithme précédent suppose, à chaque étage du treillis, de mettre à jour la métrique des chemins et de choisir parmi deux d'entre eux, celui qui en possède la plus faible. Un tel processeur réalise donc les trois fonctions élémentaires: **ajout** (*Add*), **comparaison** (*Compare*) et **sélection** (*Select*). Nous représentons les métriques dans une fenêtre coulissante de longueur $n(v+1)$, pondérée par le poids maximal d'une métrique d'entrée. (Pour le codage *soft*, ce poids est différent de 1.) C'est-à-dire qu'une métrique dépasse la taille maximale, on lui retranche la longueur de la fenêtre. Ceci n'est possible que parceque les métriques sont contenues dans la moitié de la fenêtre. Par conséquent, les métriques de chemins, sans cesse croissantes, restent bornées. Puisqu'une paire d'états bien choisie (la paire x et $x+2^{v-1}$) produit toujours une autre paire d'états (la paire $2x$ et $2x+1$), l'unité ACS peut être conçue comme une paire de processeurs. La figure 5.5 montre comment est construit un tel processeur: les métriques sont mises à jour en ajoutant la valeur des métriques de branches ($l(0x, x0)$ par exemple). Elles sont ensuite comparées dans la fenêtre coulissante (bloc *COMPARE*), et la plus petite d'entre elles est sélectionnée.

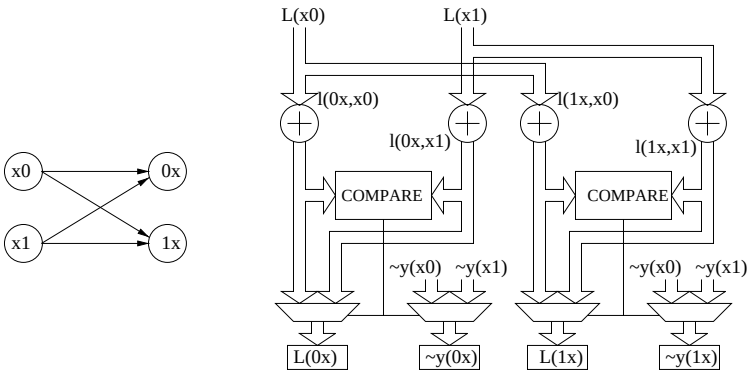

Figure 5.5: Processeur *butterfly* ACS.

Il faut à priori un processeur *butterfly* par paire d'états dans le treillis, en tout $2^{v-1} * s$ pour une séquence de longueur s. Notons que la valeur des métriques de branches (c'est-à-dire celles à ajouter aux métriques de chemins) sont calculées par un module dépendant des entrées et du treillis uniquement.

5.4.2 La gestion des survivants

Il faut également mémoriser un survivant par état. Puisqu'une longueur minimale de cinq fois la contrainte est requise pour assurer des performances acceptables, il faut au minimum $5v2^v$ points mémoires. Nous utilisons donc une RAM pour le stockage. Rader propose dans [61] d'organiser la mémoire en pointeurs décrivant du même coup les survivants: dans l'état courant est placé le bit sélectionné par l'unité ACS correspondante; celui-ci permet, par circulation, de retrouver l'état précédent. Ainsi, il suffit d'écrire un bit par état à chaque étage du treillis, mais il faut lire entièrement la mémoire au moment du décodage puisque chaque lecture renseigne le décodeur uniquement sur le prochain état le plus probable. Cette lecture, ou *trace back* s'effectue en $5v$ accès mémoire. Pour ne pas la répéter à chaque étage du treillis, nous décoderons plusieurs bits (disons p) à chaque fois. Ceci accroît la longueur de survivant requis de p mais permet de parcourir toute la mémoire et ceci tous les p étages de treillis seulement, ce qui laisse au décodeur la possibilité de répartir les accès sur plusieurs cycles d'horloge. L'utilisation de plusieurs petites RAM en parallèle permet aussi d'en accélérer l'accès.

5.4.3 La gestion des métriques

Nous avons vu que nous pouvons borner les métriques sur un nombre fini de bits. Un problème subsiste toutefois: si tous les états d'un même étage ne sont pas décodés en même temps, une double mémorisation est nécessaire puisque les métriques décodées vont prendre la place de métriques qui n'ont pas encore été exploitées. (Par exemple

pour $v = 3$, les états 0 et 4 donnent les métriques des états 0 et 1, mais l'état 1 n'a pas encore été exploité.) Rader propose donc de laisser les métriques en place: les métriques des états $0x$ et $1x$ sont placées dans les états $x0$ et $x1$ respectivement. Il n'y a plus de risque d'écrasement mais il faut effectuer une circulation pour déterminer dans quel état physique du banc de registres se trouve réellement une métrique d'un état logique. (Dans l'état physique $0x$ se trouve l'état logique $x0$). Trois gestions différentes des métriques sont utilisées pour chacune des architectures: l'architecture micro-contrôlée utilise une RAM (donc circulante) puisqu'une seule paire de métriques est requise à chaque étape. L'architecture parallèle utilise un banc de registres pour atteindre toutes les métriques d'un seul coup. L'architecture *pipelinée* ne peut utiliser une location fixe puisque l'état d'un étage peut être calculé après le même état d'un étage précédent! Nous disperserons les métriques à l'intérieur des *butterflies* et nous y accéderons ainsi sans multiplexage.

5.4.4 Organisation générale

Le décodeur s'articule autour des trois unités précédentes. Deux processus majeurs cohabitent: le parcours du treillis avec mise à jour des survivants et des métriques d'une part, et le décodage de la séquence par *trace back* d'autre part. La figure 5.6 représente cette organisation: les métriques sont stockées dans un ensemble de registres ou de RAM qui alimente les processeurs. Leurs sorties sont mémorisées dans la mémoire des survivants. D'un autre côté, celle-ci est régulièrement lue par le processus de *trace back* pour déduire la valeur la plus probable de la séquence initiale.

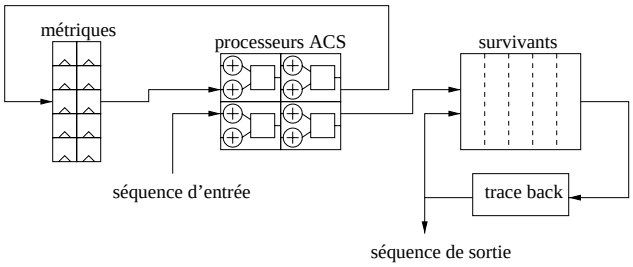

Figure 5.6: Organisation générale.

5.5 Ordonnancement des états du treillis

Nous avons souligné précédemment qu'une approche naive reviendrait à associer un processeur par paire d'états pour chaque étage de treillis. En fait, on décide d'utiliser le même processeur pour: (1) plusieurs états d'un même étage, et (2) plusieurs séquences d'étages. On obtient alors un ensemble de processeurs paramétrables dans deux dimensions: spatiale et temporelle. La figure 5.7 montre le treillis de longueur

infinie. On se limite alors à six processeurs qui seront utilisés pour traiter tous les états d'un même étage (dimension spatiale), et tous les étages (dimension temporelle). Ils pourront être connectés comme indiqué sur la figure du bas.

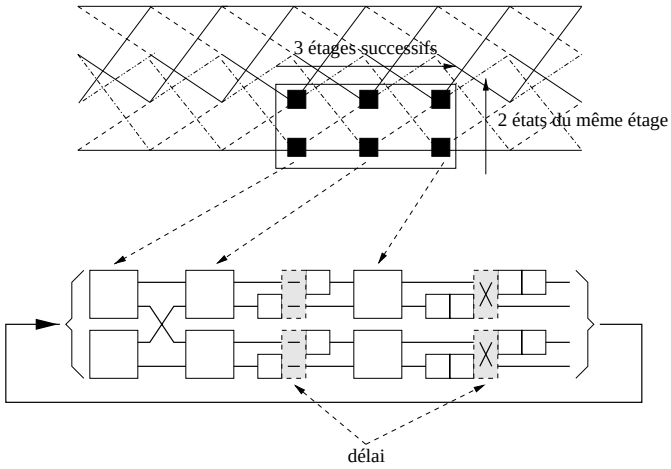

Figure 5.7: Ordonnancement des états du treillis.

Bitterlich et Meyr ont montré dans [62] que l'ordonnancement des états sur l'ensemble des processeurs était optimal lorsque les processeurs utilisaient les états dès qu'ils étaient produits. Or nous savons que les états x et $x + 2^{v-1}$ produisent les états $2x$ ou $2x + 1$, donc l'ordonnancement du tableau 5.1 utilise, à l'étage i, les états produits à l'étage $i - 1$ le cycle suivant.

étage	2^5	2^4	2^3	2^2	2^1	2^0	délai
0	.	.	.	m_0	m_1	m_2	-
1	.	.	m_0	m_1	m_2	.	-
2	.	m_0	m_1	m_2	.	.	-
3	m_0	m_1	m_2	.	.	.	1
4	m_1	m_2	.	.	.	m_0	2
5	m_2	.	.	.	m_0	m_1	4

Tableau 5.1: Ordonnancement ($v = 6$) pour 8x6 processeurs.

Cette table utilise des processeurs à un seul état de sortie (non *butterfly*). Elle peut s'étendre sans difficulté à une table de processeurs *butterflies*. Un état est représenté par la valeur des bits de poids $2^5 \cdots 2^0$. Un point signifie que 0 et 1 sont à la fois disponibles. Il y a donc 8 processeurs pour chaque étage (trois points donc 2^3 états cibles). La valeur de $m = 2^2 m_2 + 2^1 m_1 + 2^0 m_0$ représente le cycle d'ordonnancement (la référence 0 étant le premier cycle de l'étage). Par exemple, au temps $m = 2$

($m_1 = 1, m_2 = m_0 = 0$), l'étage 0 traite les états $2, 10, 18, 26, 34, 42, 50, 58, 18$ correspondant à l'état $010 m_0 m_1 m_2$. On remarque que les étages s'enchaînent sans délai excepté pour les trois derniers. En effet, un processeur a besoin des états x et $x+2^{v-1}$. Lorsque l'étage 4 commence à travailler ($m_0 = m_1 = m_2 = 0$), alors l'état produit à $m_0 = 1$ n'est pas encore disponible. Il faut donc attendre un cycle de plus. Pour les étages suivants, il faut attendre 2 et 4 cycles pour que m_1 et m_2 passent à 1 respectivement. Ces délais sont représentés sur la figure 5.6 par les lignes à retard (*FIFOs*) entre les processeurs. Il s'agit de l'architecture générique *scalable*.

De cet ordonnancement général se déduisent les deux architectures classiques de décodeur de Viterbi: lorsque la dimension spatiale est minimisée à 1 (un seul processeur par étage), l'architecture en *cascade* [63, 64] est obtenue, dont la longueur optimale est $v - 1$. Un délai exponentiel est alors inséré entre chaque étage. Lorsque la dimension temporelle est minimisée à 1, l'architecture *Trellis Pipeline Interleaving (TPI)* [65] est obtenue. Les états produits ne peuvent plus entrer dans le processeur suivant (il n'y en a plus!). Ils doivent alors attendre que les processeurs de l'étage courant aient terminé pour les réutiliser. Afin de ne pas introduire des délais inutiles mais les utiliser pour augmenter le débit, Dawid et al. [65] ont décidé de *pipeliner* les processeurs pour que les états produits puissent immédiatement re-rentrer dans un des processeurs. Dans cette architecture également, un certain nombre de délais (ou cycles *morts*) sont rajoutés. Ceux-ci correspondent aux dépendances entres données consécutives qu'il faut résoudre avant d'entrer dans le *pipeline*. Les simulations effectuées à l'aide de nos prototypes virtuels nous ont convaincus d'implanter la solution *TPI* qui possède la remarquable caractéristique d'atteindre les solutions massivement parallèles et micro-contrôlées lorsque 2^{v-1} et 1 processeurs sont respectivement utilisés.

5.6 Méthodologie de conception

Outre l'application de la méthodologie décrite dans le chapitre 6, notons toutefois plusieurs points spécifiques:

- la théorie est assez facile mais l'obtention d'une solution totalement générique nécessite d'approfondir plusieurs architectures spécifiques.
- La comparaison des architectures se fait sur la base du nombre de points mémoire requis pour stocker métriques et survivants, ainsi que sur le nombre de processeurs *butterflies*. En réalité, la complexité d'un processeur peut augmenter en fonction du degré de *pipeline*. Ce coût réel sera pris en compte au niveau comportemental.
- L'implantation RTL permet non seulement d'affiner des solutions mais aussi d'obtenir des solutions nouvelles qui ne peuvent être atteintes par le comporte-

mental. Ceci est dû à la nécessité d'ordonnancer les états sur les processeurs. La synthèse comportementale n'est pas capable de définir automatiquement des ensembles disjoints d'états pour permettre l'ajout de *pipeline*. La solution comportementale rejoint l'architecture TPI mais sans le *pipeline*: le débit est donc plus faible.

Nous fournissons le volume des parties cognitives, de prototype C et matérielles. Les critères quantitatifs du tableau 5.2 sont les nombres d'articles ou de livres étudiés, le nombre de ligne de code C et VHDL respectivement. Le temps de conception est donné en nombre de semaines pour un homme. Le temps de l'implémentation RTL est en grande partie dû à l'architecture TPI, beaucoup plus difficile à concevoir que les autres.

description	critères quantitatifs	volume	temps (sem)
cognitive	livres	1	0.5
	articles	9	2.5
prototype C	lignes de code	2877	
	lignes de commentaire	1649	
	total	4526	3
matérielle RTL	lignes de code	2943	
	lignes de commentaire	1754	
	total	4697	11
matérielle comportementale	lignes de code	505	
	lignes de commentaire	307	
	total	812	3

Tableau 5.2: Volume des différentes descriptions Viterbi.

5.7 Résultats expérimentaux du décodeur de Viterbi

Les résultats expérimentaux obtenus sont le coût et les performances des architectures, pourvu que toutes les fonctionnalités soient remplies. Pour que le décodeur soit totalement configurable, les trois architectures implantées acceptent comme paramètres:

- la contrainte v du code convolutionnel,
- les polynômes générateurs,
- le nombre de sortie N,
- le nombre de bits de codage *soft*,

- la longueur des survivants,
- la répartition du *trace back* donc du débit.

Nous avons représenté sur le schéma 5.8 les performances des trois architectures, La contrainte est de 8 pour une longueur de survivants variant de 48 à 63, et 4 bits d'entrée sont utilisés. Les ronds représentent le prototype virtuel, les carrés le code RTL, et les étoiles le code comportemental. Il montre que les prototypes virtuels envisageaient un meilleur débit: ceci provient du temps d'accès aux RAM de stockage qui empêche d'augmenter la fréquence d'horloge en même temps de le *pipeline*. Ensuite il montre que l'espace de la surface et du débit sont bien recouverts par les trois solutions: l'architecture TPI propose un bon compromis entre les deux solutions extrêmes micro-contrôlées et parallèles. Finalement, remarquons que la description comportementale est moins bonne que la description RTL. La description comportementale permet de parcourir toutes solutions entre la solution parallèle et micro-contrôlée. Nous avons obtenu beaucoup d'autres solutions à partir de ce code, mais dont les performances se situent entre des points déjà représentés. Toutefois, puisque le *pipeline* n'y est pas inséré en fonction de l'ordonnancement des états, le débit diminue progressivement. La solution TPI, par contre, parcours ces mêmes solutions en conservant un débit presque optimal puisque le *pipeline* diminue le chemin critique. Nous présentons donc cette comparaison montrant les limites de la conception comportementale face à la précision du RTL.

5.8 Architecture Reed-Solomon à haut débit

Toutes les architectures implémentées utilisent les mêmes algorithmes. Toutefois, les contraintes de surface et de débit peuvent engendrer divers degrés de parallélisme. Nous présentons ici l'architecture de haut débit qui sera ensuite dégénérée en architecture dense ou micro-contrôlée. Afin de réduire la connectique, l'architecture haut débit est également systolique.

5.8.1 Encodage

L'encodage s'effectue comme tout code cyclique en multipliant l'information $i(x)$ et le polynôme générateur $g(x)$. Pour obtenir un code systématique, les mots du code sont construis à l'aide de l'opération suivante:

$$i(x).x^{2t} = q(x).g(x) + r(x) \Longrightarrow c(x) = i(x).x^{2t} + r(x). \tag{5.3}$$

En effet, le mot $c(x)$ est bien multiple de $g(x)$, et différents mots d'information conduisent à différents mots du code puisque les symboles de poids forts sont justement les symboles d'information. L'encodage sytématique est parfaitement réalisé par un LFSR [66]. Les coefficients du polynôme générateur pondérés par le terme

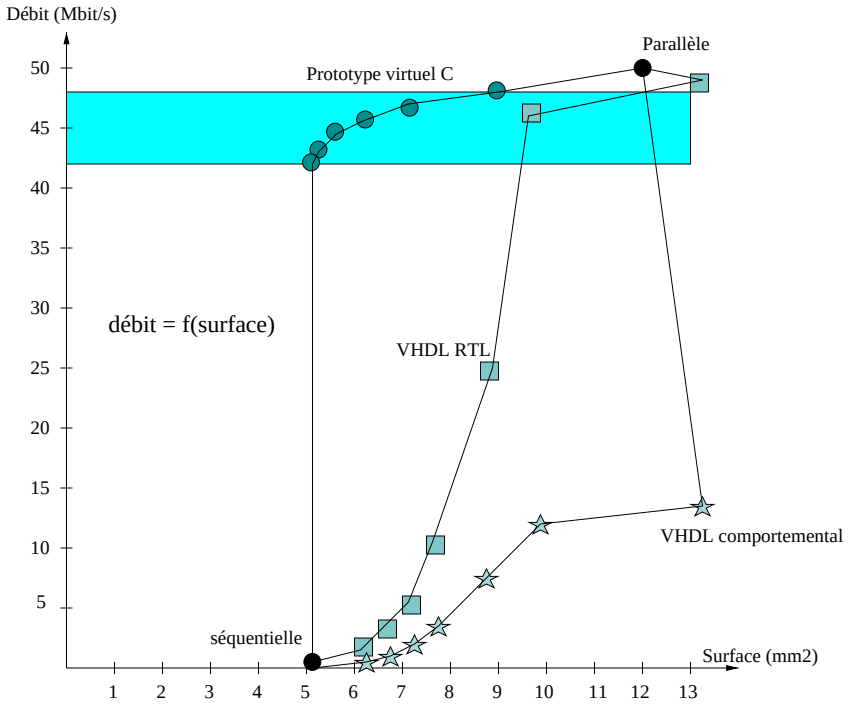

Figure 5.8: Performance des architectures de Viterbi.

de plus haut degré réalise la division, tandis que l'addition du symbole information réalise la multiplication par x^{2t}. La figure 5.9 présente l'architecture du codeur ou les registres et opérateurs sont à base de corps de Galois.

L'information est directement transmise en sortie pendant que le LFSR calcule la redondance. Lorsque les k symboles ont été transmis, la division est achevée. Le signal *Debut-I* passe alors à 0 tandis que le LFSR est à la fois décalé est progressivement initialisé. Dès la redondance transmise, il est près à effectuer l'encodage suivant [67].

5.8.2 Architecture de décodeur

L'algorithme naif de PGZ (Peterson-Gorenstein-Zierler) [54] ne convient pas à une implémentation VLSI à cause de la résolution des systèmes linéaires. Par conséquent, le décodeur est composé de quatre blocs majeurs: le calcul des syndromes, l'expansion des polynômes effacements, la génération des polynômes locateur et évaluateur d'erreur, et l'évaluation de ces polynômes. Pendant la latence du décodeur, une FIFO retarde les symboles pour les ajouter à l'erreur. Un diagramme est fourni en figure 5.10 où apparaît la latence de chacun des blocs. Le calcul des syndromes doit attendre jusqu'au dernier symbole reçu, la division euclidienne exige une latence de $6t$ pour qu'un polynôme de longueur $2t$ traverse sériellement $2t$ unités de division, et

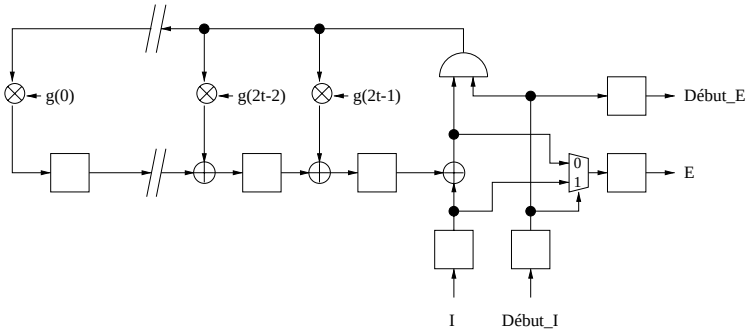

Figure 5.9: Architecture du codeur.

l'algorithme de Berlekamp-Massey peut se faire en $2t$ cycles mais exige alors une fréquence d'horloge double.

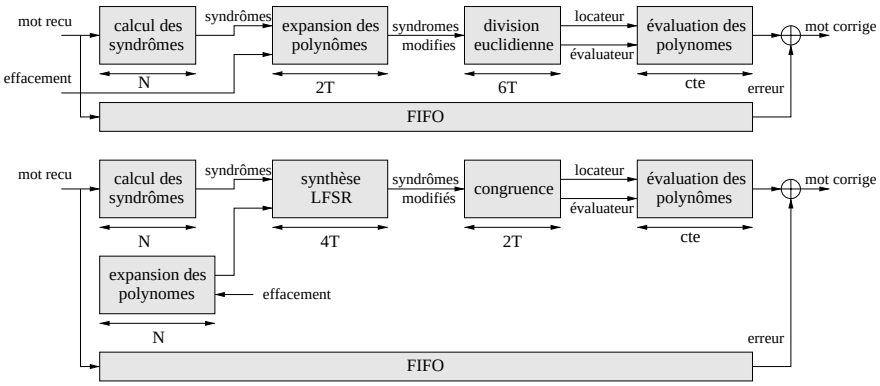

Figure 5.10: Diagramme du décodeur.

5.8.3 Calcul des syndromes

Le calcul des syndromes S_j est une évaluation du mot reçu de longueur N en $2t$ racines du polynôme générateur: $S_j = w(\alpha^{a+j})$ pour $1 \leq j \leq 2t$. On reçoit les symboles de poids forts en premier (de w_{N-1} à w_0). Pour éviter l'exponentiation, des multiplications itératives sont effectuées à chaque symbole reçu et les syndromes sont graduellement calculés selon la règle de Horner:

$$S_j = ((\cdots(w_{N-1}\alpha^{a+j}) + w_{N-2})\alpha^{a+j} + \cdots + w_1)\alpha^{a+j} + w_0. \tag{5.4}$$

Le but est non seulement de remplacer l'exponentiation par une multiplication mais aussi de fournir une décomposition récursive du calcul des syndrômes. Si S_j est initialisé à 0 alors S_j contient le syndrome quand le symbole w_0 est reçu:

$$S_j \longleftarrow S_j \alpha^{a+j} + w_i. \tag{5.5}$$

dont l'implémentation systolique est donnée en figure 5.11. Elle consiste en $2t$ cellules identiques, chacune composée d'un additionneur et d'un multiplieur par une constante.

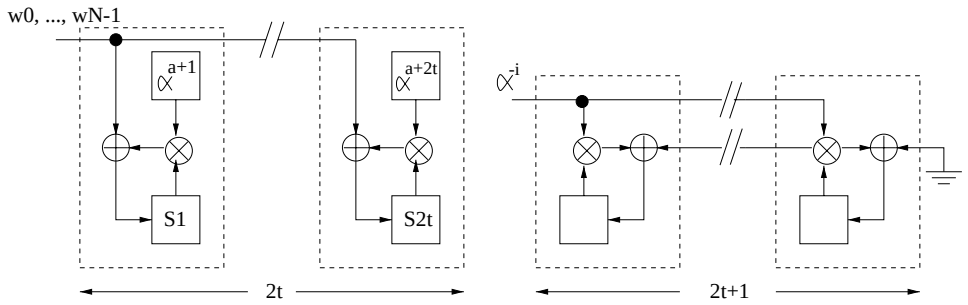

Figure 5.11: Un réseau systolique de calcul des syndromes et d'expansion de polynômes.

5.8.4 Equation clé

Soit les polynômes locateur $\sigma(x)$ et évaluateur $\omega(x)$ suivants:

$$\begin{cases} \sigma(x) & = \prod_{i=0}^{e-1}(1 - u_i X) \\ \omega(x) & = \sum_{i=0}^{e-1} y_i u_i \prod_{j=0 \neq i}^{e-1}(1 - u_j X) \end{cases} \tag{5.6}$$

Notons que $\sigma(x)$ et $\omega(x)$ sont premiers entre eux. $S(x)$ étant le polynôme dont les coefficients sont les premiers $2t$ syndromes, les polynômes locateur et évaluateur verifient l'équation clé suivante:

$$\omega(x) = \sigma(x)S(x) mod x^{2t}. \tag{5.7}$$

Puisque deux algorithmes majeurs existent pour résoudre cette équation clé, nous présentons ces deux algorithmes que nous avons intensivement implémentés.

5.8.5 La division euclidienne

Le probème, introduit dans [68], consiste à trouver une paire de polynôme $\sigma'(x)$ et $\omega'(x)$ de degrés inférieurs ou égal à t et $t-1$ respectivement, verifiant l'équation clé (5.7). Alors $\sigma'(x)\omega(x) = \sigma(x)\omega'(x)$ puisque les degrés de $\sigma'\omega$ et $\sigma\omega'$ sont inférieurs à $2t$. Nous en déduisons que $\sigma'(x) = \mu(x)\sigma(x)$ et $\omega'(x) = \mu(x)\omega(x)$ où $\mu(x)$ est un polynôme approprié. Si, de plus, $\sigma'(x)$ et $\omega'(x)$ sont premiers entre eux, alors $\sigma'(x) = \sigma(x)$ et $\omega'(x) = \omega(x)$ à une constante près. L'algorithme d'Euclide appliqué à $S(x)$ et x^{2t} fournit les polynômes consécutifs $r_i(x)$ obtenus selon:

$$r_{i-2}(x) = q_i(x)r_{i-1}(x) + r_i(x) \tag{5.8}$$

avec $r_{-1}(x) = x^{2t}$ et $r_0(x) = S(x)$. Si on définit les polynômes $U_i(x)$ et $V_i(x)$ en utilisant la même division:

$$\begin{cases} U_i(x) & = q_i(x)U_{i-1}(x) + U_{i-2}(x) \\ V_i(x) & = q_i(x)V_{i-1}(x) + V_{i-2}(x) \end{cases} \tag{5.9}$$

avec $U_0(x) = 1$, $U_{-1}(x) = 0$, $V_0(x) = 0$ et $V_{-1}(x) = 1$, alors le polynôme reste $r_i(x)$ à la $i^{ième}$ itération peut être exprimé en fonction de r_0 et r_{-1} selon:

$$r_i(x) = (-1)^i[-V_i(x)r_{-1}(x) + U_i(x)r_0(x)]. \tag{5.10}$$

On continue les itérations jusqu'à ce que les restes satisfassent $deg(r_{k-1}) \geq t$ et $deg(r_k) < t$ si bien que les polynômes $\omega'(x) = (-1)^k\gamma r_k(x)$ et $\sigma'(x) = \gamma U_k(x)$, où γ est une constante qui rend $\sigma'(x)$ unitaire, satisfont l'équation clé avec $deg(\omega') < t$ et $deg(\sigma') \leq t$. En utilisant le fait que $U_i(x)V_{i-1}(x) - U_{i-1}(x)V_i = (-1)^i$, nous prouvons que ω' et σ' sont premiers entre eux et sont donc les solutions du problème.

Parce que des quotients sont requis dans la forme usuelle de l'algorithme d'Euclide, des circuits d'inversion (en fait $2t$) dans corps de Galois doivent être conçus. Pour éviter cette inversion coûteuse et pour trouver les quantités $r_i(x)$, $\gamma_i(x)$ et $\lambda_i(x)$ verifiant $\gamma_i(x)x^{2t} + \lambda_i S(x) = r_i(x)$, Shao à developpé dans [69][70] les relations suivantes:

$$\begin{cases} r_0(x) = x^{2t}, & r_i(x) & = [\sigma_{i-1}b_{i-1}r_{i-1}(x) + \bar{\sigma_{i-1}}a_{i-1}q_{i-1}(x)] \\ & & -x^{|l_i-1|}[\sigma_{i-1}a_{i-1}q_{i-1}(x) + \bar{\sigma_{i-1}}b_{i-1}r_{i-1}(x)] \\ \lambda_0(x) = 0, & \lambda_i(x) & = [\sigma_{i-1}b_{i-1}\lambda_{i-1}(x) + \bar{\sigma_{i-1}}a_{i-1}\mu_{i-1}(x)] \\ & & -x^{|l_i-1|}[\sigma_{i-1}a_{i-1}\mu_{i-1}(x) + \bar{\sigma_{i-1}}b_{i-1}\lambda_{i-1}(x)] \\ \gamma_0(x) = 1, & \gamma_i(x) & = [\sigma_{i-1}b_{i-1}\gamma_{i-1}(x) + \bar{\sigma_{i-1}}a_{i-1}\eta_{i-1}(x)] \\ & & -x^{|l_i-1|}[\sigma_{i-1}a_{i-1}\eta_{i-1}(x) + \bar{\sigma_{i-1}}b_{i-1}\gamma_{i-1}(x)] \\ q_0(x) = S(x), & q_i(x) & = \sigma_{i-1}q_{i-1}(x) + \bar{\sigma_{i-1}}r_{i-1}(x) \\ \mu_0(x) = 1, & \mu_i(x) & = \sigma_{i-1}\mu_{i-1}(x) + \bar{\sigma_{i-1}}\lambda_{i-1}(x) \\ \eta_0(x) = 0, & \eta_i(x) & = \sigma_{i-1}\eta_{i-1}(x) + \bar{\sigma_{i-1}}\gamma_{i-1}(x) \end{cases} \tag{5.11}$$

où a_{i-1} et b_{i-1} sont les coefficients de tête de $r_{i-1}(x)$ et $q_{i-1}(x)$ respectivement, $l_{i-1} = deg(r_{i-1}(x)) - deg(q_{i-1}(x))$ et $\sigma_{i-1} = 1$ si $l_{i-1} \geq 0$, $\sigma_{i-1} = 0$ si $l_{i-1} < 0$. L'indéterminé σ_{i-1} peut être supprimée et les équations réduites en échangeant les polynômes r_{i-1}/q_{i-1}, λ_i/μ_{i-1}, γ_i/η_i avant calcul si $l_{i-1} < 0$. Ces équations fournissent un couple de polynômes solutions mais peuvent exiger plus d'itérations que la méthode originale, qui sont dans tous les cas limitées par $2t$.

5.8.6 Implantation systolique de la division euclidienne

Brent et Kung ont à la fois souligné l'importance des architectures systoliques [71] qui procurent une structure régulière, parallèle, *pipelinée* et facilement interconnectable appropriée à l'implémentation VLSI, mais ont aussi proposé un réseau de $2t$ cellules systoliques effectuant une division euclidenne [72]. Les polynômes entrent dans la cellule sériellement et sont décalés de telle sorte que les termes de tête sont toujours alignés. Ceux-ci sont alors mémorisés pour que les multiplications et additions s'appliquent successivement sur les autres coefficients (*réduction*). L'alignement des polynômes est possible en faisant circuler le polyôme divisé plus rapidement que les autres. La condition pour laquelle une division soit possible est que le terme de tête du diviseur ne soit pas nul. Si tel est le cas, le degré de celui-ci est simplement décrémenté pour que le terme suivant soit considéré à la prochaine unité (*décrémentation*). Lorsque le degré du polynôme $r(x)$ est plus petit que celui de $q(x)$, on échange les paires r/λ et q/μ. La fonctionnalité peut donc être décrite par le diagramme de flot suivant, montrant les cinq chemins d'opérations possibles: décalage, réduction, réduction et échange, décrémentation, décrémentation et échange. De ce diagramme, présenté en figure 5.12, nous avons conçu le circuit suivant: les polynômes entrent sériellement, signalés par un signal *début* accompagné des degrés de $r(x)$ et $q(x)$. L'état de la machine d'états finis est le même jusqu'au signal *début* suivant. C'est alors que les trois conditions *cond1/2/3* décident du choix du prochain état. Puisque les actions peuvent être appliquées aux termes de tête lors du changement d'état (par exemple de la fin d'un décrémentation à une réduction), les actions sont associées aux transitions. La machine d'états finis de Mealy est donnée à droite du diagramme correspondant.

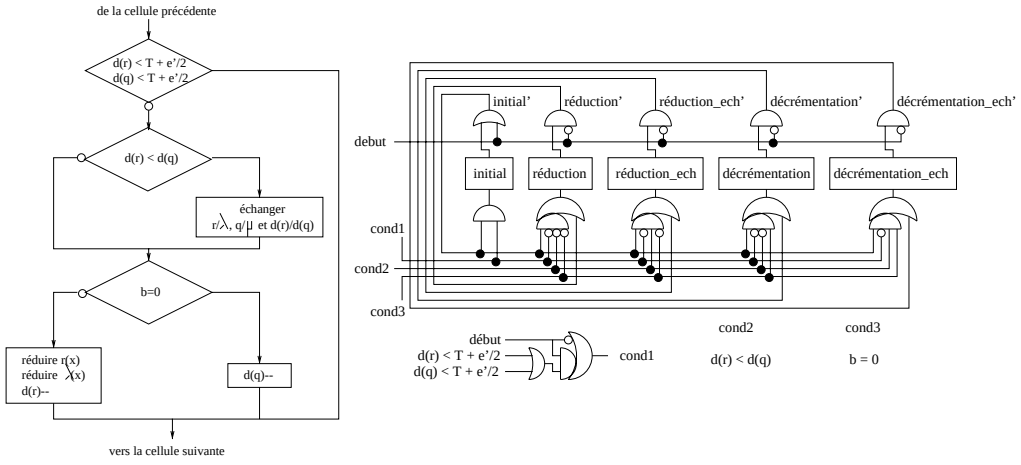

Figure 5.12: Diagramme de flot et machine d'états finis de la division.

Le chemin de données correspondant à la figure 5.13 effectue en fait neuf actions:

outre les cinq précédentes, les quatre opérations associées aux transitions *début* vers une des quatre actions sont ajoutées. Les registres sont alors contrôlés par un multiplexeur 5:1 (transitions *initial*, *réduction(-ech)*, *décalage(-ech)*, *initial→réduction(-ech)* et *initial→réduction(-ech)*). Un signal *fin* marquant la fin des polynômes traverse la cellule à la même vitesse que les polynômes associés afin de pouvoir les extraire sans difficulté ni erreur. La division complète consiste donc à mettre bout-à-bout $2t$ de ces cellules bien que les dernières ne seront souvent utilisées que comme simples registres à décalage. Nous avons conçu cette cellule de telle sorte que soit minimisé le nombre d'opérateurs, multiplieurs notamment, et de points mémoire afin d'en réduire la surface. La caractéristique systolique de ces cellules les rend parfaitement interconnectable lors de la phase de placement routage.

5.8.7 Synthèse de LFSR

L'équation clé peut se réecrire

$$\frac{\omega(x)}{\sigma(x)} = S_1 + S_2 x + S_3 x^2 + \cdots \tag{5.12}$$

avec $\sigma(x)$ et $\omega(x)$ premiers entre eux et le degré de $\sigma(x)$ plus grand que celui de $\omega(x)$. Appelons les coefficients du LFSR (*Linear Feedback Shift Register*) le polynôme connexion $C(D) = 1 + c_1 D + \cdots + c_L D^L$, comme illustré dans la figure 5.14.

Par conséquent, $\sigma(x)$, pondéré d'un facteur constant requis pour que $\sigma(0) = 1$ est l'unique polynôme connexion du plus petit LFSR qui génère la séquence des syndromes [73]. Comme le nombre d'erreurs corrigibles est au plus t, les premiers $2t$ syndromes sont suffisants pour produire la solution de longueur minimale.

Soit $L_N(S)$ la longueur minimale de tous les LFSR qui génère $S_0, S_1, \cdots, S_{N-1}$. Si une LFSR de longueur $L_N(S)$ génère $S_0, S_1, \cdots S_{N-1}$ mais pas $S_0, S_1, \cdots S_{N-1}, S_N$, alors $L_{N+1}(S) = max[L_N(S) + N + 1 - L_N(S)]$. Pour le prochain syndromes S_N,

$$S_j + \sum_{i=1}^{L_N} c_i S_{j-1} = \begin{cases} 0, & j = 0, 1, \cdots, N-1, \\ d_N & j = N. \end{cases} \tag{5.13}$$

Si l'écart d_N entre S_N et le symbole généré par le LFSR n'est pas nul, alors un nouveau LFSR doit être construit. Soit M la longueur de la séquence avant le dernier changement de polynôme connexion. Massey démontre que le polynôme $C(D) = C^{(N)}(D) - \frac{d_N}{d_M} D^{N-M} C^{(M)}(D)$ est un choix satisfaisant pour $C^{(N+1)}(D)$. L'algorithme suivant permet de trouver le polynôme connexion.

Algorithme 1 - Synthèse de LFSR

1. $C(D) \leftarrow 1, B(D) \leftarrow 1, x \leftarrow 1, L \leftarrow 0, b \leftarrow 1, n \leftarrow 0$.
2. si $n = N$, stop. Autrement calculer $d = S_N + \sum_{i=1}^{L} c_i S_{N-i}$.

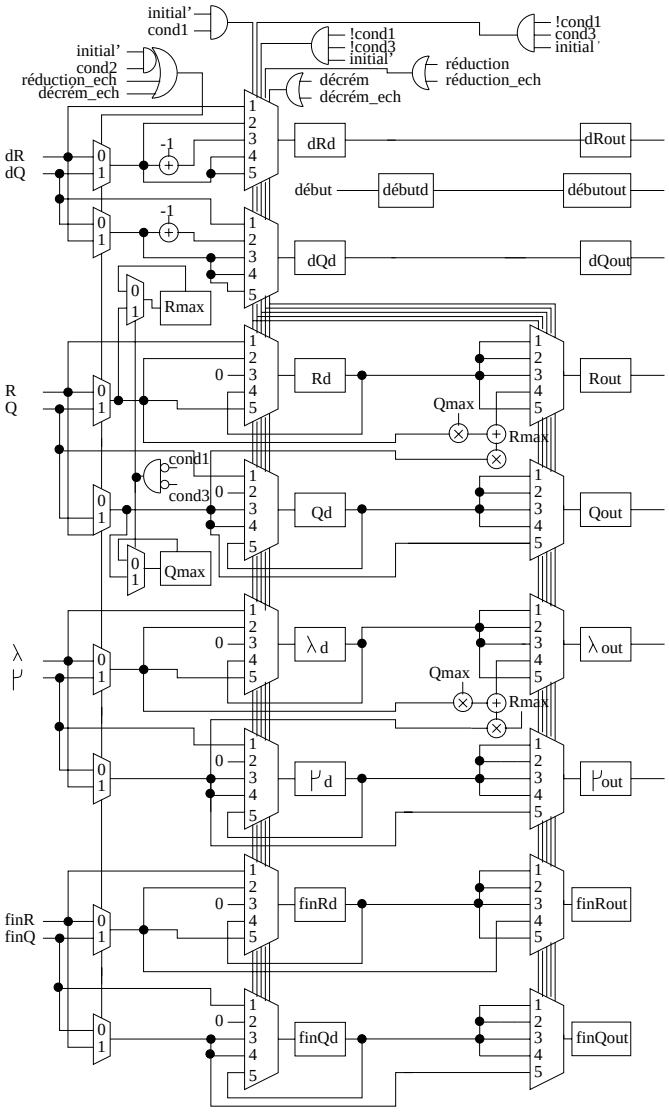

Figure 5.13: Chemin de données de la cellule de division.

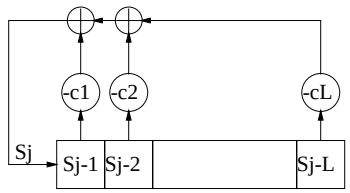

Figure 5.14: Polynôme connexion.

3. si $d = 0$, alors $x \leftarrow x + 1$ et aller en 6.

4. si $d \neq 0$ et $2L > N$ alors $\begin{cases} C(D) \leftarrow C(D) - \frac{d}{b} D^x B(D) \\ x \leftarrow x + 1. \end{cases}$ et aller en 6.

5. si $d \neq 0$ et $2L > N(?)$ alors $\begin{cases} T(D) \leftarrow C(D), \\ C(D) \leftarrow C(D) - \frac{d}{b} D^x B(D), \\ L \leftarrow N + 1 - L, \\ B(D) \leftarrow T(D), \\ b \leftarrow d, \\ x \leftarrow 1. \end{cases}$ et aller en 6.

6. $N \leftarrow N + 1$ et aller en 2.

Une fois le polynôme locateur trouvé, on en déduit le polyôme évaluateur par l'équation clé.

5.8.8 Implantation de la synthèse de LFSR

Le circuit implantant la synthèse de LFSR est donné par Massey en figure 5.15. Il nécessite $4t$ multiplications, $4t$ additions et une inversion. Comme le chemin critique passe à travers trois multiplieurs et $log_2(2t) + 1$ additionneurs, nous avons séparé la logique supérieure et inférieure en introduisant des registres d et b et en doublant la latence de décalage des polynômes. Les opérations à effectuer sont les suivantes:

- si $d = 0$, décaler B et S d'une position.
- si $d \neq 0$ et $2L > N$, mettre le commutateur en position 1 et décaler B et S d'une position.
- si $d \neq 0$ et $2L \leq N$, mettre le commutateur en position 2 et
  $\begin{cases} remplacer & b\ par\ d, \\ remplacer & L\ par\ N + 1 - L, \\ décaler & B\ et\ S\ d'une\ position, \\ charger\ un\ 1 & dans\ le\ premier\ étage\ de\ B. \end{cases}$

Lorsque la logique inférieure est active, les registres S, B, C, b et L sont inchangés et d et b^{-1} sont chargés. Alors que lorsque la logique supérieure est active, les registres S, B, C, b et L sont chargés.

5.8.9 Décodage d'effacements

Le décodage des effacements est important puisqu'il est deux fois plus facile que le décodage d'une erreur si bien qu'une erreur peut être remplacée par deux effacements.

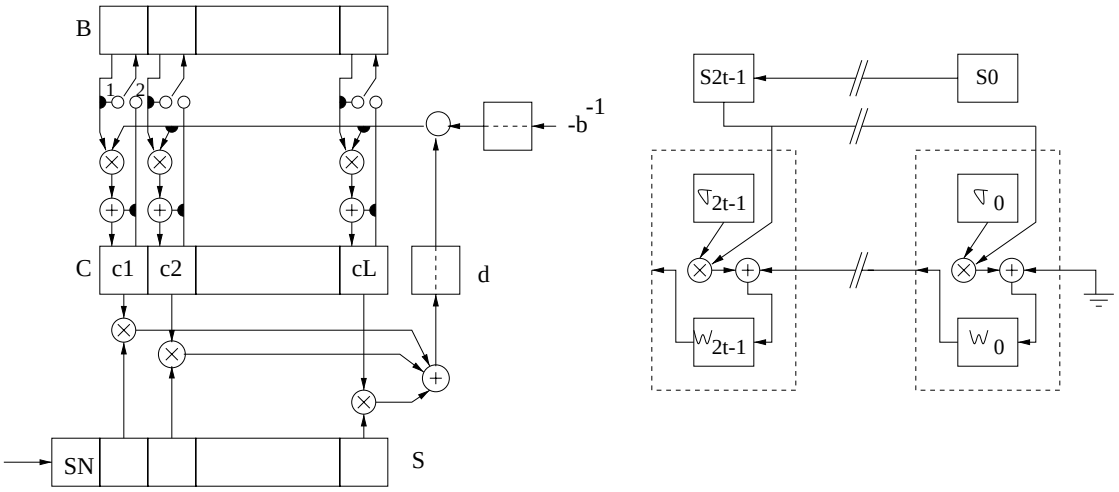

Figure 5.15: Chemin de données de la synthèse de LFSR et du calcul de congruence.

Les deux algorithmes précédents peuvent être adaptés au décodage des effacements (les erreurs sont bien sûr toujours décodées). Le polynôme d'effacement est:

$$\Lambda(x) = \prod_{i=0}^{e'-1} (u_{i+e}x + 1), \tag{5.14}$$

avec e et e' le nombre d'erreurs et d'effacements respectivement. $\sigma(x)$ reste le polynôme locateur mais le polynôme évaluateur devient:

$$\omega(x) = \sum_{i=0}^{e+e'-1} y_i u_i \prod_{j=0 \neq i}^{e+e'-1} (1 - u_i x). \tag{5.15}$$

Pour la division euclidienne, Shao [70] forme le polyôme syndromial modifié

$$T(x) = S(x).\Lambda(x) mod x^{2t}. \tag{5.16}$$

L'équation clé devient

$$\omega(x) = \sigma(x)T(x) mod x^{2t} \tag{5.17}$$

avec $deg(\omega(x)) < e+e'$ et $deg(\sigma(x)) = e+e'$. En appliquant la division euclidienne à $T(x)$ et x^{2t} jusqu'à ce que le reste satisfasse $deg(r_{k-1}) \geq \frac{2t+e'}{2}$ et $deg(r_k) < \frac{2t+e'}{2}$ et puisque le degré du polynôme locateur est limité par la différence entre le degré de départ $2t$ et la condition d'arrêt $\frac{2t+e'}{2}$ soit $\frac{2t-e'}{2}$, les deux polyômes satisfont l'équation clé (5.17) avec des degrés adéquats. Le polynôme locateur $\sigma(x).T(x)$ peut-être directement obtenu par l'algorithme de division en initialisant $\mu_0(x)$ à $\Lambda(x)$.

Pour la synthèse de LFSR, l'algorithme est modifié comme suit:

- initialiser les polynômes B et C avec le polynôme d'effacement,
- remplacer n et L par $n - e'$ et $L - e'$ respectivement dans le test de longueur. Toutefois, n va toujours jusqu'à N.

Le registre C contient alors le polynôme locateur $\sigma(x).T(x)$.

Pour les deux algorithmes, le nombre d'itérations diminue lorsque le nombre d'effacements augmente. Afin de garder une latence constante, nous en effectuerons toujours $2t$ sachant que les dernières peuvent se comporter comme de simples registres à décalage.

5.8.10 Expansion des polynômes

Pour corriger les effacements, un polynôme effacement et syndromial modifié doivent être calculés, possédant un signal d'effacement et les $2t$ syndromes. $T(x)$ et $\Lambda(x)$ sont calculés comme le produit des effacements successifs $x - \alpha^{-i}$ par $S(x)$ et 1 respectivement, parceque le produit $p(x).(x - \alpha^{-i}) = x.p(x) - \alpha^{-i}p(x)$ peut-être implanté par le réseau systolique donné en figure 5.11, initialisé à $p(x)$. Comme précédemment, les polynômes expansés sont sériellement extraits.

5.8.11 Relation de congruence

L'unité de congruence a pour but de calculer le polynôme évaluateur en fonction des polynômes syndromial et locateur, selon l'équation clé (5.7). Ceci est réalisé par le réseau systolique donné en figure 5.15, multipiant $\sigma(x)$ par les syndromes successifs et accumulant le résultat partiel dans $\omega(x)$. Bien entendu, les termes plus grands que x^{2t} sont éliminés.

5.8.12 Evaluation des polynômes

Possédant les polynômes locateur et évaluateur, les locations et amplitudes des erreurs doivent être extraites. Alors que les locations sont les racines α^{-loc} de $\sigma(x).T(x)$, les amplitudes sont données grâce à la dérivée de $\sigma(x)$ [74]:

$$y_i = \alpha^{-a.pos} \frac{\omega(\alpha^{-loc})}{\sigma'(\alpha^{-loc})} \tag{5.18}$$

Chien propose dans [75] d'évaluer $\sigma(x)$ par des décalages circulaires pour détecter les racines avec peu de coût matériel. Nous réarrangeons le circuit, fourni à gauche de la figure 5.16, pour évaluer tous les types de polynômes en multipliant la valeur du terme x^i en α^{-pos} par α^i pour déterminer la valeur du même terme en $\alpha^{-(pos-1)}$.

Les polynômes ont des degrés variant de $2t$ (locateur avec effacements) à $t-1$ (évaluateur sans effacements), donc requiert un nombre variable d'unités d'évaluation. La première position est évaluée à l'initialisation en multipliant les coefficients de $\sigma(x)$, $\omega(x)$ ou $\sigma'(x)$ par $(\alpha^{-deg})^N$. Ensuite, les valeurs précédentes sont réactualisées par la récursion $val \leftarrow val.\alpha^{deg}$. Un arbre logarithmique d'additionneurs permet d'évaluer tous les termes. L'algorithme de Forney permet alors de calculer l'amplitude de l'erreur, qui n'est ajoutée que si la valeur du polynôme locateur en cette position est nulle. La figure 5.16 de droite implante l'équation 5.18: à condition que $\sigma(\alpha^{-loc})$ soit nul, on inverse $\sigma'(\alpha^{-loc})$ pondéré par la constante $\alpha^{-a.pos}.\omega(\alpha^{-loc})$, et on ajoute cette erreur au mot reçu w, retardé dans la *FIFO*.

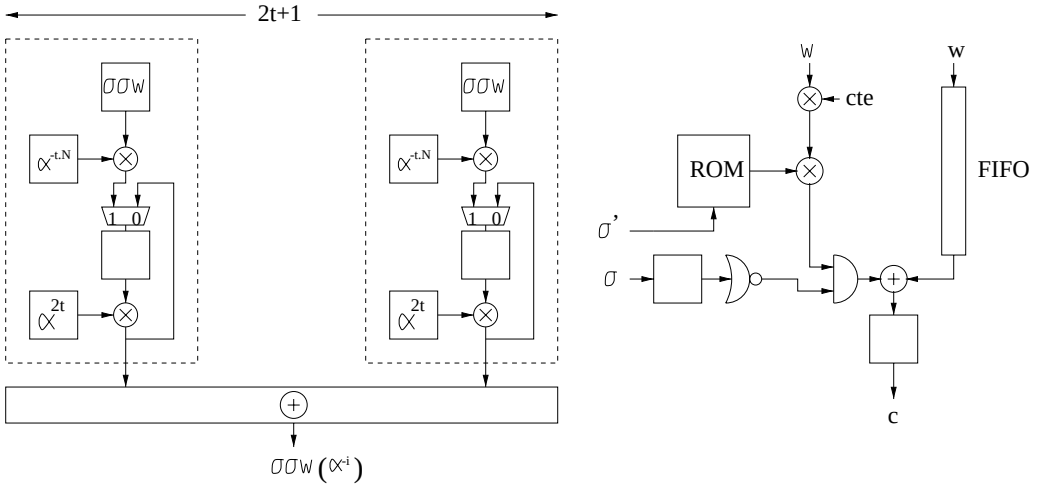

Figure 5.16: Un circuit pour évaluer les polynômes.

5.8.13 Détection d'échec de décodage

L'échec de décodage arrive lorsque le mot reçu n'appartient à aucune sphère de décodage et donc le mot décodé ne pourra pas faire partie du code. Le décodeur doit donc le signaler au reste du circuit. Pour les polynômes verifiant l'équation clé (5.7), une condition nécessaire et suffisante pour détecter un échec est démontrée par Sawarte [76]:

- $deg(\omega(x)) \geq deg(\sigma(x))$ ou
- $\sigma(x)$ n'a pas $deg(\sigma(x))$ racines distinctes ou
- au moins une amplitude est nulle ou n'appartient pas au corps des symboles.

Par conséquent, si aucune de ces caractéristiques n'est vérifiée, alors le mot décodé est toujours un mot du code.

5.8.14 Arithmétique dans le corps de Galois

L'addition et la multiplication dans le corps de Galois sont simplement réalisées comme la somme et le produit de polynômes modulo le polynôme primitif décrivant le corps de Galois [77]. Toutefois, il est possible d'obtenir un réseau systolique pour la multiplication [78][79] ou bien de simplifier la réalisation au prix d'un changement de base de représentation du corps de Galois [80][81]. De même, nous avons réalisé l'inversion par une ROM. Il est possible de concevoir un circuit combinatoire d'inversion: en changeant de base [82] ou en utilisant un algorithme spécifique tel la division euclidienne [83] ou le pivot de Gauss [84]. Toutes ces architectures ont l'avantage de fournir une implantation régulière mais nécessite une latence importante (jusqu'à $7m$ cycles pour l'inversion).

5.9 Architectures Reed-Solomon dérivées

Le décodeur de Reed-Solomon s'insère toujours dans un système de stockage ou de transmission de données et hérite donc de contraintes très diverses. Cela peut-être le débit (produit de la latence par la fréquence d'horloge), la latence (lorsque la fréquence d'horloge est fixée) ou la surface. Nous proposons donc des solutions dégénérées de la précédente pour fournir des débits ou des latences adaptés à la surface.

5.9.1 Architecture à latence réduite

Les blocs de la figure 5.10 montrent que la latence du calcul des syndromes et de l'évaluation des polynômes est déjà minimale puisqu'elle dépend de l'entrée sérielle des symboles. Par contre, l'expansion et l'obtention des polynômes solutions peuvent-être optimisées car c'est le circuit qui gère la circulation des données entrantes (polynôme syndromial, polynôme effacement ou polynôme syndromial modifié). Nous diminuons la latence de la manière suivante:

- Les syndromes rentrent parallèlement dans l'unité d'expansion (ou en série par paquets de p) si bien que les $2t$ expansions successives (ou p) sont, à présent, réalisées dans le même cycle d'horloge: la surface et le chemin critique sont augmentés d'un facteur $2t$ (ou p).
- Le polynôme syndromial modifié n'est plus sériellement introduit dans la première unité de division mais tous les coefficients sont réduit dans le même cycle. De plus, les $2t$ (ou p) divisions peuvent être effectuées dans le même cycle si bien que la surface et le chemin critique sont augmentés de $2t$ (ou p).

La surface augmente donc localement d'un facteur p, la fréquence d'horloge diminue d'un facteur p mais la latence devient:

$$latence = N + \frac{4t}{p} + 2 \tag{5.19}$$

Il s'agit de la solution idéale d'un circuit de communication où la fréquence d'horloge est assez basse pour permettre de chaîner les opérations combinatoires.

5.9.2 Architecture à surface réduite

La solution massivement parallèle précédente peut-être implantée par un architecture micro-contrôlée. Le principe est le suivant: toutes les opérations combinatoires sont effectuées par la même unité arithmétique et logique (UAL), supprimant ainsi plusieurs centaines de multiplieurs et additionneurs. La surface séquentielle reste la même puisqu'il faut toujours stocker les données intermédiaires entre deux opérations. Toutefois, le stockage du micro-code s'effectue dans une ROM et les données sont placées dans une RAM pour diminuer la surface des points de mémorisation équivalents et pour ne pas avoir à concevoir le circuit de décodage d'accès aux données.

Le micro-programme est écrit en langage C, puis translaté en langage machine sur des lignes de 28 bits. Il se compose de deux sortes d'opérations:

1. les opérations combinatoires ou séquentielles. Elles nécessitent deux opérandes gauche et droite, une opérande résultat et une opération. les opérandes sont des éléments du corps de Galois, un bit, ou un degré (entier). Elle peuvent être stockées dans des registres, la mémoire ou des constantes cablées. Les opérations sont des opérations de corps de Galois (addition, nultiplication, inversion, ..), des opérations logiques (*Et*, *Ou*, comparaison, ..) ou des opérations entières (incrémentation, décalage, ..). Elles s'effectuent en deux cycles: le premier joue le rôle de *fetch* en allant chercher les opérandes; le deuxième effectue l'opération et stocke le résultat comme précisé dans le micro-code. Le temps d'accès des mémoires ne permet pas en effet d'effectuer tout le travail dans le même cycle.

2. les contrôles de flot permettent d'effectuer des opérations conditionnelles (saut) ou itératives (boucle). Les données à tester doivent être obligatoirement des registres ou des constantes si bien qu'un seul cycle est nécessaire. L'UAL est plus simple puisque seules les opérations logiques élémentaires sont effectuées. l'adresse de la micro-instruction suivante est maintenant fonction du résultat du test précédent.

Le schéma logique est fourni en figure 5.17. Elle présente le traitement d'une opération combinatoire (deux cycles) et d'une opération de contrôle de boucle (un cycle). La ROM contient 252 micro-instructions de 28 bits de largeur, la RAM contient $N + 32t + 31$ lignes de largeur m ($m = 8$ en général) et la latence du décodage est de $(N + 4t + 2)(275 + 232t)$ cycles d'horloge dans le pire cas, c'est-à-dire qu'un symbole

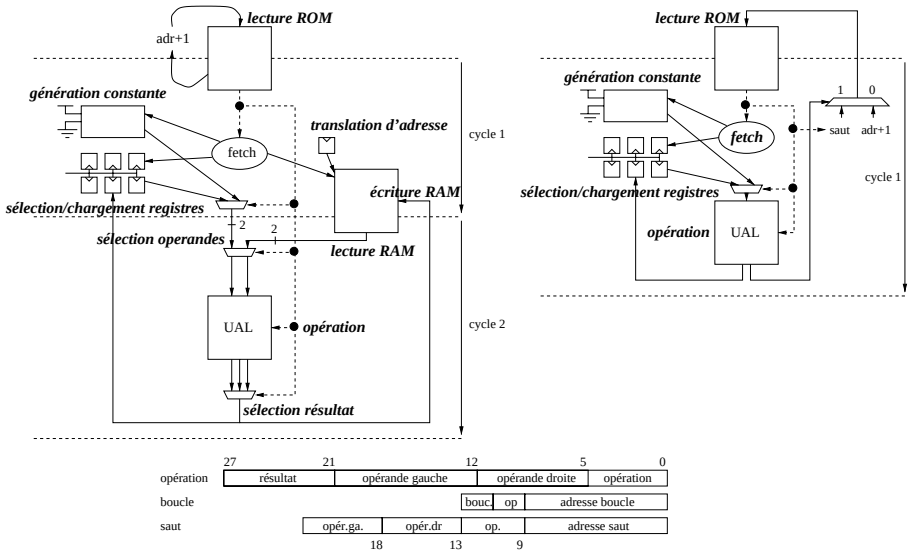

Figure 5.17: Schéma de l'architecture micro-contrôlée.

est corrigé tous les $275 + 232t$ cycles. Bien que petite, la RAM occupe la moitié de la surface totale. La coût de cette architecture est, bien entendu, le débit qui chute désormais à:

$$débit = \frac{F * m}{275 + 232t}. \tag{5.20}$$

où F est la fréquence d'horloge.

5.9.3 Architecture mixtes

Les architectures précédentes effectuent les opérations soit parallèlement, soit en séquence, ce qui engendre deux solutions de débit et de surface extrême. Il est judicieux de remarquer qu'il existe un autre degré de liberté quant à l'allocation des opérations: la plupart des données à traiter sont des polynômes (syndromial, locateur, évaluateur) si bien qu'il est possible, dans chaque unité de traitement, d'effectuer les opérations sur tous les coefficients en même temps ou séquentiellement. En d'autres termes, nous avons déjà réalisé des architectures totalement parallèles et totalement séquentielles, mais nous pouvons aussi choisir de traiter séquentiellement les polynômes dans une architecture parallèle ou de traiter parallèlement les polynômes dans une architecture séquentielle. On obtient alors deux architectures mixtes dont le débit est diminué ou augmenté localement d'un facteur $2t$.

La solution séquentielle avec traitement des polynômes parallèles est obtenue en plaçant les mémoires les coefficients des polynômes de telle manière à pouvoir les

accéder dans le même cycle. L'UAL est bien sûr plus grosse de $2t-1$ multiplieurs et additionneurs mais les autres opérations restent les mêmes et les boucles sont supprimées. Le schéma de principe des opérations combinatoires et séquentielles est donné en figure 5.18, dont le fonctionnement est identique à l'architecture micro-contrôlée précédente.

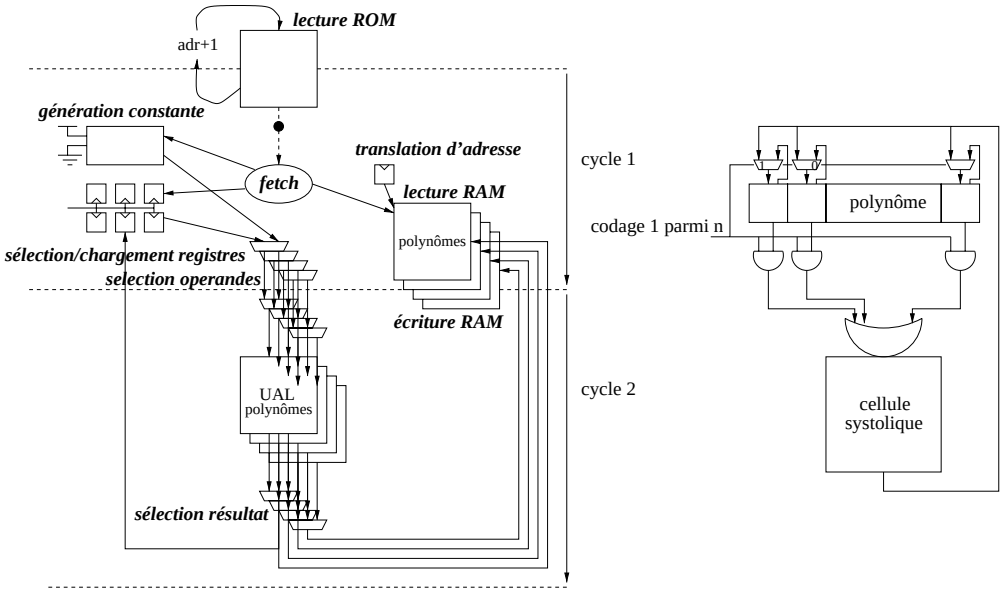

Figure 5.18: Architectures mixtes.

La solution parallèle avec traitement séquentiel des polynômes est obtenue en n'utilisant qu'une seule cellule systolique par unité de calcul, de sélectionner les coefficients, d'effectuer l'opération et de réactualiser les coefficients séquentiellement. Le principe de l'architecture est fourni dans la même figure. On voit que $2t$ cellules sont remplacées par une cellule munie d'un arbre de sélection d'opérandes.

Notons finalement que nous avons encore dégénéré ces architectures pour réaliser des décodeurs de code BCH (Bose-Chaudhuri-Hocquenheim) qui ont la particularité de transmettre des bits à la place de symboles. L'algorithme de décodage est simplifié. En particulier, les codes corrigeant une, deux ou trois erreurs peuvent être construits en déterminant directement des syndromes la position des erreurs. Nous ne nous étendrons pas plus sur ces codes.

5.10 Résultats expérimentaux des codes RS

Pour le code comportemental, un de nos soucis était d'une part d'expérimenter l'outil de synthèse comportementale en mode cycle fixé. D'autre part, l'algorithme initial

est inexploitable sans fournir des détails architecturaux: l'évaluation et les opérations polynomiales sont en effet prohibitifs. Ces deux arguments nous ont incité à utiliser également le comportemental pour décrire une architecture précise. Toutefois, alors que le RTL nous fournit une architecture systolique, nous choisissons pour le comportemental une architecture sans contrainte de connexions donc plus dense. La différence réside dans le module de division euclidienne: au lieu d'utiliser $2t$ modules systoliques effectuant une étape de division sur un coefficient de polynôme à la fois, nous utiliserons une seule unité de division effectuant l'opération sur tous les coefficients simultanément. Le coût des architectures avant implantation est estimé par le nombre de multiplications et de registres pour polynômes.

Nous présentons le volume des parties cognitives, de prototypes C et matérielles. Les critères quantitatifs sont les nombres d'articles ou de livres étudiés, le nombre de lignes de code C et VHDL respectivement (tableau 5.3). Le temps de conception est encore calculé en nombre de semaines pour un homme. Le volume RTL est bien supérieur au volume comportemental puisque nous avons conçu plusieurs architectures en RTL (massivement parallèle, dense, micro-contrôlées) contre une seule en comportemental.

description	critères quantitatifs	volume	temps (sem)
cognitive	livres	3	4
	articles	19	6
prototype C	lignes de code	5097	
	lignes de commentaire	2143	
	total	7240	6
matérielle RTL	lignes de code	11393	
	lignes de commentaire	5265	
	total	16658	24
matérielle comportementale	lignes de code	1387	
	lignes de commentaire	581	
	total	1968	6

Tableau 5.3: Volume des différentes descriptions Reed-Solomon.

Les résultats expérimentaux obtenus sont le coût et les performances des architectures, pourvu que toutes les fonctionnalités soient remplies. Pour que le décodeur soit totalement configurable, les trois architectures implantées acceptent comme paramètres:

- la dimension du corps de Galois,
- le polynôme générateur,
- la longueur du code,

- la capacité de correction,
- le support des effacements.

Les performances sont obtenues sur un code de longueur 73, corrigeant 10 octets d'erreur ($m = 8$). La compexité est le nombre de portes équivalentes (*Nand2*), et la surface est la surface de silicium incluant les connexions sur la technologie 0.35μ de ST. Tout ceci est résumé dans le tableau 5.4. La surface et donc le temps de synthèse sont d'autant plus grands que le débit est important.

architectures	performances		synthèse
massivement parallèle	fréquence	$= 100$ MHz	12 heures
	complexité	$= 62K$	
	surface	$= 3.8mm^2$	
	débit	$= 100m = 800$ MBits/s	
	latence	$= N + 6t + 10 = 143$	
dense	fréquence	$= 100$ MHz	3 heures
	complexité	$= 38K$	
	surface	$= 2.4mm^2$	
	débit	$= 100m/(2t + 1) = 48$ MBits/s	
	latence	$= (N + 6t + 7)(2t + 1) = 2940$	
micro-contrôlée	fréquence	$= 100$ MHz	40 min
	complexité	$= 5K$	
	surface	$= 1.0mm^2$	
	débit	$= 100m/(275 + 232t) = 0.30$ MBits/s	
	latence	$= (N + 4t + 2)(275 + 232t) = 298425$	

Tableau 5.4: Résultats expérimentaux du Reed-Solomon.

5.11 Conclusion

Nous avons présenté la structure des blocs complexes de codage correcteur d'erreur les plus fréquemment utilisés. De la théorie et de l'état de l'art conséquent, nous avons conçu des blocs pour les principaux critères d'optimisation possibles (surface et débit), allant de l'architecture micro-contrôlée à l'architecture massivement parallèle. Comme nous le verrons plus en détail dans le chapitre suivant, tous ces blocs possèdent leurs propres fiches techniques et sont commercialement proposés comme IP's.

Chapitre 6

Les méthodologies de réutilisation

6.1 Critères d'efficacité d'une méthode de réutilisation

6.1.1 L'utilisation

Pout être réutilisé, un bloc doit avant tout être **utilisable**. Les principaux critères de qualité d'un bloc utilisable sont bien connus: la qualité et la clarté du code commenté, les scripts de synthèse et de simulation (*test benches*), ainsi qu'une documentation complète. Seulement alors faudra-t-il penser à exporter le bloc et le rendre aussi attractif que possible.

6.1.2 La distance cognitive

L'efficacité d'une méthode de réutilisation se mesure par le critère intuitif qu'est la distance cognitive [2]. Elle représente l'effort intellectuel requis pour l'utiliser.

L'**abstraction** est la caractéristique essentielle d'une méthode de réutilisation puisque sans elle, le concepteur serait obligé de dégager lui-même la fonctionnalité, les caractéristiques et le mode d'intégration des blocs qu'il possède. Par conséquent, une bonne abstraction revient à abaisser la distance cognitive. La seconde caractéristique est **l'automatisation**: elle consiste à utiliser des compilateurs afin de translater la spécification du macro-bloc en sa réalisation. Elle apporte rapidité de conception et fiabilité, et abaisse donc également la distance cognitive.

Comme nous l'avons vu au second chapitre, le monde de la synthèse est constitué d'abstraction et d'automatisation. Par conséquent, le matériel est la cible idéale des méthodes de réutilisation. Afin de quantifier cette distance cognitive, nous mettons à disposition trois critères d'évaluation que sont la sélection, la spécialisation et l'intégration.

6.1.3 La sélection

Une méthodologie de blocs réutilisables aide les concepteurs à localiser, comprendre, comparer et sélectionner les blocs dont ils ont besoin. Si les caractéristiques d'une IP sont bien connues de son concepteur, celles-ci ne le sont, pour les autres, que par l'intermédiaire des informations abstraites qu'il aura pris soin de diffuser.

L'émergence du réseau mondial (*Web*) permet de cataloguer les blocs suivant une taxonomie uniforme et de les diffuser sur internet. Cette taxonomie guide le concepteur dans sa recherche jusqu'à obtention d'un groupe précis de solutions. Une fois les blocs **localisés**, le concepteur doit **comprendre** les caractéristiques de chacune d'entre eux: les algorithmes d'implantation doivent faire référence à l'état de l'art le plus récent. Cette précaution assure que la solution implantée possède le consentement implicite de la communauté scientifique. Les caractéristiques doivent être suffisamment explicites pour pouvoir **comparer** les différentes solutions d'un même algorithme. Ces comparaisons se font sur les critères d'optimisations étudiés au chapitre précédent (fréquence, surface, consommation) mais aussi sur les performances de l'architecture: latence, débit, et sur la fiabilité de la solution proposée. Ne reste plus qu'à **sélectionner** le composant en fonction des critères de comparaison précédents.

Afin de satisfaire à ce besoin de sélection, le concepteur doit donc non seulement décrire de manière abstraite les caractéristiques de son composant dans des fiches de données (*data sheet*) suffisamment étoffées mais aussi assurer leur diffusion.

6.1.4 La spécialisation

Puisque les besoins des concepteurs sont tous différents alors qu'il n'existe pas de composant spécifique pour chaque application, les composants similaires sont rassemblés dans un même composant générique. Le concepteur doit donc effectuer le travail inverse qui consiste à **spécialiser** une solution générique en une solution spécifique en fixant la partie variable de la spécification. Les outils de spécialisation sont, entre autres:

- la paramétrisation d'une architecture avant synthèse qui fournit donc une instance compilée spécifique,
- la configuration d'une instance compilée générique,
- les contraintes de synthèse qui permettent, à partir d'une même description abstraite, d'optimiser un des critères de sélection.

Le concepteur du composant doit donc penser aux besoins des utilisateurs en proposant une solution à la fois générique et spécialisable. Reprenons l'exemple du

décodeur Reed-Solomon précédent. Tous les paramètres du code et de l'architecture peuvent être librement fixés par le concepteur: le polynôme primitif P sur lequel se construit le corps de Galois $GF(2^m)$, sa dimension m, la longueur du code N, la capacité de correction T, la correction d'effacement $Eras$, l'algorithme (synthèse de LFSR ou division euclidienne), l'architecture utilisée (micro-contrôlée, parallèle, mixte), l'utilisation de blocs durs ou non pour l'implantation de la FIFO et de l'inversion. Toutes ces options sont figées avant synthèse de telle sorte qu'un composant spécialisé, donc performant, est synthétisé. La partie fixe est réduite au profit d'une complète partie variable. Les détails architecturaux qui sont donc la partie cachée de l'IP ont été choisis de telle sorte qu'ils optimisent les performances du composant, la latence par exemple. Le schéma 6.1 montre comment on peut obtenir trois réalisations totalement différentes à partir de la même spécification. Nous obtenons alors trois codecs d'architecture, de taille et de capacité différentes.

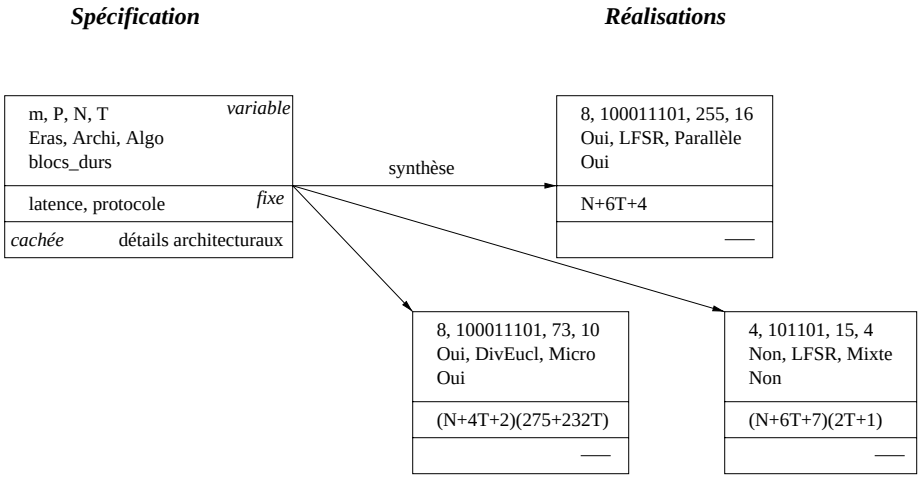

Figure 6.1: Spécialisation d'une IP.

6.1.5 L'intégration

Une fois son composant taillé sur mesure, le concepteur doit combiner une collection de composants dans un même système matériel. Un langage d'interconnexion permet typiquement d'exporter les composants des entités qui les implantent jusqu'aux entités qui les utilisent en faisant communiquer les composants dans un même niveau d'abstraction. Par exemple, une description en VHDL RTL permet au concepteur d'instancier chaque composant en utilisant les outils de synthèse classiques. Ensuite, pour **intégrer** le composant, le concepteur doit comprendre son interface, c'est-à-dire le protocole de communication avec le monde extérieur, et l'insérer dans le reste du système. Par exemple, l'intégration d'une carte vidéo d'un ordinateur de type PC se

fait immédiatement par *plug-and-play* parce-que les ports et le protocole de communication de la carte sont normalisés et compris de tous (système et autres composants): les connecteurs PCI assurent l'harmonisation du langage de communication, et le contrôleur de bus PCI assure la compatibilité des protocoles.

Le concepteur doit donc faciliter l'intégration de son composant en normalisant le langage de description, et en fournissant précisément le protocole de communication. Les descriptions des codecs (codeur et décodeur) Reed-Solomon et Viterbi sont effectuées à la fois en code comportemental et RTL. Ces derniers sont synthétisables sur plusieurs outils commerciaux (Synopsys, Mentor) et sont intégrables par simple instantiation de composant. En particulier, nous fournissons l'ensemble des scripts de synthèse qui permettent:

- d'effectuer la compilation de tout le composant en deux passes: une première passe synthétise les blocs internes selon les contraintes du concepteur et en utilisant les contraintes temporelles moyennes pour interfacer le bloc avec le monde extérieur. Dans une deuxième passe, chaque bloc est caractérisé et un jeu de contraintes précises est cette fois appliqués aux blocs précédents, ce qui permet d'optimiser tous les composants.

- d'accepter toutes les contraintes du concepteur que ce soit la surface, la fréquence de fonctionnement ou bien l'effort d'optimisation.

- de choisir n'importe qu'elle cible technologique: la seule indication d'une bibliothèque FPGA permet par exemple d'effectuer une prototypage du composant.

Le schéma de la figure 6.2 illustre graphiquement l'enchaînement logique des trois critères précédents. La distance cognitive est donc l'effort qu'il faut faire pour passer de la spécification à la réalisation via les étapes de sélection, spécialisation et intégration.

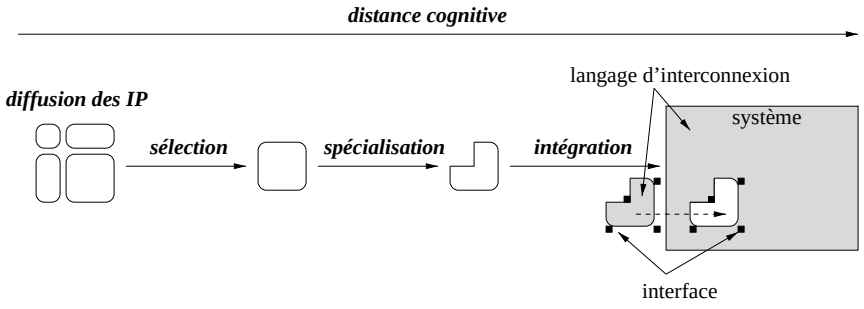

Figure 6.2: Critères d'efficacité d'une méthode de réutilisation.

6.2 Techniques de réutilisation

Les techniques de réutilisation sont les méthodes dont le concepteur dispose pour intégrer dans son circuit un bloc réutilisable. Il ne s'agit nullement d'une méthode de conception de l'IP mais simplement comment favoriser leur utilisation. La réutilisation de bloc consiste à intégrer une macro déjà conçue de quelque niveau de complexité que ce soit. Le niveau d'abstraction de la description peut aller du comportemental au bloc placé et routé.

La méthodologie de conception structurée, introduite par Mead et Conway [85], repose sur les concepts de **hiérarchie** et **régularité** [86]: la hiérarchie permet de manier des blocs de petite taille tandis que la régularité accroît les possibilités d'identifier un bloc déjà conçu. La méthodologie suivante permet non seulement d'utiliser des blocs, mais aussi d'enrichir la bibliothèque de composant.

6.2.1 Le partitionement

Le **partitionement**, appliqué à la spécification, divise le système en plusieurs blocs plus simples. On introduit donc une hierarchie qui isole les fonctionnalités à concevoir ou réutiliser indépendemment du reste du circuit. Un bon partitionnement laisse les sous-systèmes indépendants les uns des autres, ne laissant visible que leur interface. Le choix du partitionement peut aussi être guidé par la disponibilité de blocs spécifiques (approche ascendante). Le coût d'une solution peut se mesurer à la complexité des blocs qui ne pourront être réutilisés. Il s'agit donc de trouver le meilleur recouvrement du système sur la bibliothèque de composants virtuels dont on dispose déjà.

6.2.2 La réutilisation

La conception de chaque sous-système dépend de l'existence d'un bloc de même fonctionnalité en bibliothèque. S'il en existe un, il faut le **réutiliser** pour une conception en un temps constant. Si non, il faut le concevoir en un temps fonction de la complexité du bloc; le lecteur est alors renvoyé au paragraphe 6.3. La réutilisation consiste à sélectionner le bloc dans une bibliothèque de composants, à le spécialiser en enlevant toute fonctionnalité ou performance superflue, puis à l'intégrer. Les détails architecturaux du module étant cachés, seul l'interface le lie au reste du système.

6.2.3 L'abstraction

L'**abstraction** consiste à tirer profit des conceptions effectuées afin d'enrichir la bibliothèque de composants. Le temps passé sur la macro servira donc à plusieurs utilisations. L'abstraction extrait les informations nécessaires à la synthèse, à l'intégration,

à la sélection, etc...

La méthodologie est illustrée en figure 6.3: la lecture de la bibliothèque correspond à l'étape intermédiaire de réutilisation. L'étape amont de partitionement favorise l'obtention de blocs déjà existants en bibliothèque, et l'étape en aval d'abstraction garde avantage des conceptions effectuées. Lorsque la bibliothèque ne fournit pas de blocs pouvant réaliser un des blocs spécifiés, il faut le concevoir.

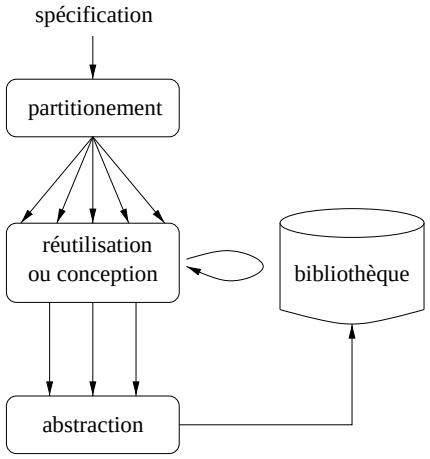

Figure 6.3: Méthodologie de réutilisation.

6.3 Méthodologies de conception

Nous présentons ici les principales méthodes de conception d'un bloc qui peut être ou non réutilisé. Outre les modèles classiques disponibles dans le manuel de réutilisation [87], nous proposons une technique supplémentaire qui nous a permis de concevoir l'ensemble des blocs de codage (chapitre 6).

6.3.1 Le modèle en cascade

Le modèle en **cascade** (*waterfall*) suppose qu'une équipe soit associée à un niveau d'abstraction donné. Lorsqu'une phase est achevée, la suivante peut alors commencer sans jamais revenir à une phase précédente: un expert propose une spécification en fonction des exigences de fonctionnalité qui lui sont imposées. Dans un deuxième temps, l'équipe de conception code cette spécification en effectuant les verifications fonctionnelles. Ensuite, l'équipe de synthèse effectue la compilation du bloc jusqu'à obtention d'un réseau de portes verifiant les contraintes temporelles. Dans un quatrième temps, le bloc est placé et routé pour obtenir un prototype fabriqué et testé. Le risque majeur de cette approche est que lorsque le circuit ne satisfait pas toutes

les contraintes, il faut réitérer l'ensemble du flot. La convergence vers une solution acceptable peut ainsi prendre trop de temps. La figure 6.4a illustre le caractère séquentiel de cette approche. Par conséquent, on s'oriente aujourd'hui vers un modèle parallèle.

6.3.2 Le modèle en spirale

Le modèle en **spirale** met en concurrence l'ensemble des équipes que sont: les concepteurs logiciels et matériels, l'équipe de synthèse et de placement et routage. L'élaboration du bloc avance seulement lorsque toutes les équipes sont en accord sur la solution adoptée. Il ne faut donc jamais recommencer tout le flot de conception lorsqu'une contrainte n'est pas respectée. La figure 6.4b présente cette approche: le modèle en spirale gère simultanément les domaines fonctionnels, matériels, temporels et physiques jusqu'à obtention d'une solution satisfaisant toutes les contraintes.

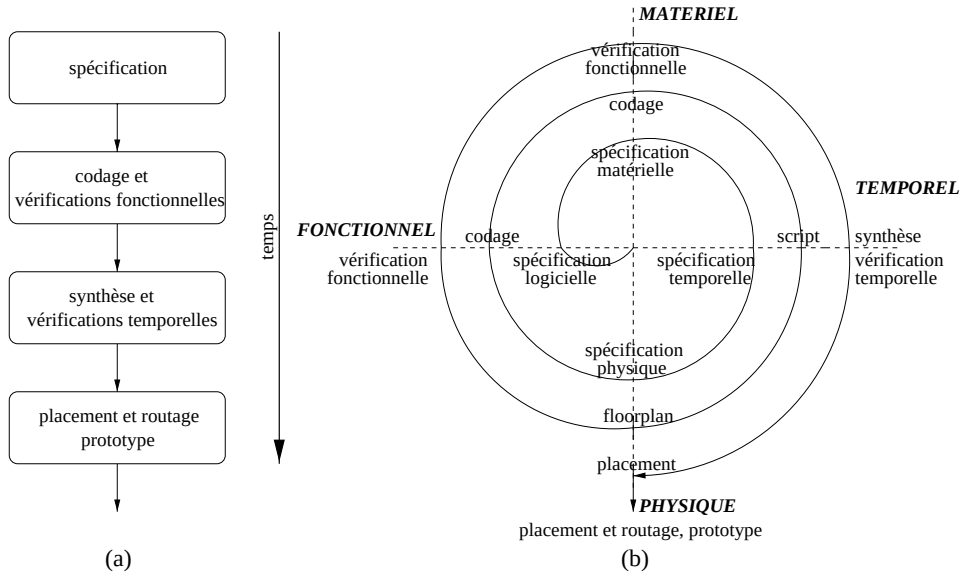

Figure 6.4: Modèles en cascade et en spirale.

6.3.3 Le modèle descendant et ascendant

Le modèle naif **descendant** consiste à spécifier un système puis à réaliser les blocs qui le composent. Il suppose donc que tous les blocs spécifiés peuvent être conçus. Malheureusement, s'il s'avère en cours de route qu'un des blocs n'est pas réalisable dans les contraintes requises, tout le processus de spécification doit être reconsidéré. Par conséquent, une meilleure méthode de conception consiste à mélanger les approches descendantes et **ascendantes** en concevant d'abord les blocs critiques qui ne

peuvent être obtenus par réutilisation. S'ils sont effectivement réalisables, on peut alors choisir une spécification les incluant.

6.3.4 Le modèle par itération

Le modèle par **itération** construit le circuit par itérations successives jusqu'à une solution acceptable. Une seule équipe réalise une première passe du cycle de conception, de la spécification jusqu'au prototypage, le plus rapidement possible. Le but est multiple: d'abord, avoir une vue d'ensemble du circuit et connaître les parties critiques; ensuite, connaître l'impact exact d'un choix architectural ou algorithmique sur les performances du circuit. Puisque cette première réalisation ne respecte certainement pas les spécifications, l'équipe peut alors corriger et raffiner la première réalisation jusqu'à obtention de la solution finale. Cette approche très naturelle consiste à débroussailler rapidement le circuit pour mieux en comprendre les points délicats et les améliorer progressivement. Sur ces quatre modèles classiques de conception nous en apportons un cinquième, qui nous a servi à concevoir les IPs de codage.

6.4 Fiabilité de la réalisation

6.4.1 Position du problème

Malgré l'efficacité constatée des méthodologies précédentes, celles-ci présentent un risque quant à la fiabilité du code produit. L'écriture du code VHDL lors du passage de la spécification (cognitive) à la réalisation (matérielle) n'est pas exempte d'erreurs. Les risques d'erreur se regroupent en deux catégories: celles qui dépendent d'une mauvaise spécification et celles qui proviennent d'un mauvais codage. La spécification peut-être erronée parce qu'incomplète ou innovante. Par conséquent, le concepteur est confronté au dilemme suivant: faut-il repenser l'architecture ou au contraire réécrire une partie du code matériel ?

L'explication de ce problème est simple: la distance cognitive entre la spécification et le codage est trop grande puisque deux étapes importantes doivent être maitrisées: la validation de l'architecture et la production du code matériel correspondant. La figure 6.5 illustre la distance entre le couple spécification/réalisation précédent.

6.4.2 Utilisation de l'abstraction

Comme présenté précédemment, l'abstraction est la meilleure technique dont on dispose pour abaisser la distance cognitive. Les deux extrémités du couple spécification/réalisation de la figure 6.5 étant imposées, utiliser l'abstraction consiste à raffiner ce couple en insérant un niveau d'abstraction intermédiaire.

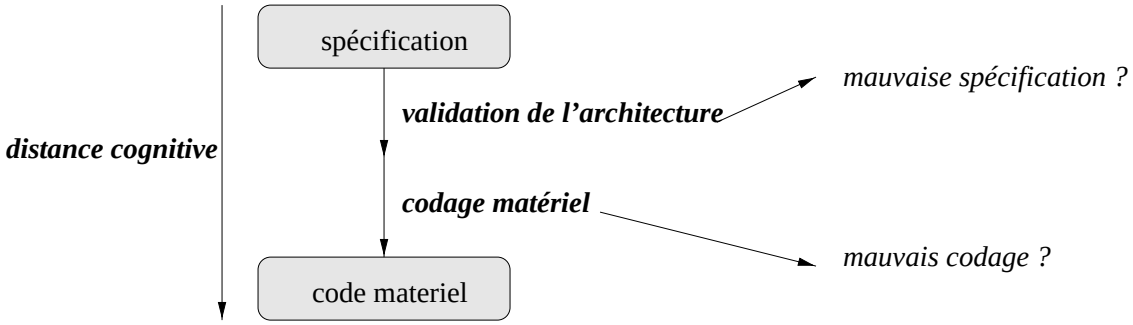

Figure 6.5: Position du problème.

Nous introduisons donc l'étape de validation dont l'objectif est d'isoler la compréhension et la validation de la spécification du codage. La validation est donc la réalisation de la spécification cognitive, mais aussi la spécification du codage matériel. Cette solution permet de découpler la validation du codage, donc de réduire la distance cognitive entre les deux couples spécification/réalisation consécutifs comme illustré en figure 6.6.

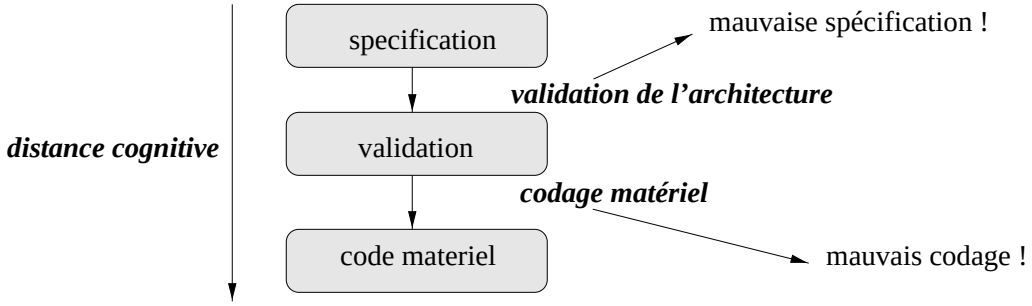

Figure 6.6: Utilisation de l'abstraction.

6.4.3 Modèle de validation

La définition de l'étape de validation nécessite la création d'un modèle de validation. Remarquons que quelque soit le langage utilisé, il subsistera toujours une incertitude quant à la provenance d'une erreur au cours de la validation: provient-elle de la spécification ou de l'écriture du modèle de validation. On ne peut pas l'exclure puisque la réalisation introduit toujours un langage de description. Toutefois, l'écriture d'un modèle de validation en langage C apporte les qualités suivantes:

- Un code C de haut niveau est d'abord écrit. La grammaire du langage C est simple par rapport au langage VHDL, la quantité de lignes de code est faible (quelques dizaines de lignes) et les détails d'architectures sont ignorés.

- Des raffinements successifs sont apportés jusqu'à obtention d'un modèle de validation pour une architecture spécifique.
- Les outils de simulation et de débuggage permettent une mise au point rapide et un nombre de stimuli plusieurs milliers de fois plus important que pour une simulation matérielle.

La validation étant découplée du codage, une erreur de simulation provient nécessairement de l'architecture. Nous somme donc amenés à proposer une description algorithmique et plusieurs descriptions architecturales. Le but de la description algorithmique n'est pas de fournir un code rapide mais une réalisation fonctionnellement correcte de l'algorithme. La meilleur façon d'y arriver est de ne pas introduire d'optimisations supplémentaires. Par contre, en tant que réalisation, l'étape de validation permet de décrire fidèlement une architecture. Elle contient tous les détails architecturaux qui doivent être codés en VHDL. En particulier, elle:

- décrit les fonctions combinatoires VHDL en langage C,
- alloue autant de variables que de points mémoire matériels,
- séquence les opérations par des boucles infinies simulant les fronts d'horloge.

Le modèle de validation est donc rythmé par la même succession de cycles d'opérations combinatoires et de mémorisation que l'est l'architecture matérielle.

A titre d'exemple, nous fournissons un pseudo-code C décrivant le calcul des syndrômes. La première description, algorithmique, utilise une procédure de forte complexité mais très simple à mettre en œuvre pour le calcul de chacun des syndrômes. La seconde description, architecturale, introduit la même fonction que celle contenue dans l'architecture parallèle: elle est plus complexe à décrire et s'effectue récursivement à chaque cycle d'horloge.

description algorithmique	description architecturale

synd = calloc(2t, sizeof(int))
for(all-synd k) {
  synd(k) ← proc-iterative(k);
}
 	
	synd = calloc(2t, sizeof(int))
	for(all-cycles clk) {
	  for(all-synd k) {
	  synd(k) ← proc-recursive(synd(k), α^k);
	  }
	}

Tableau 6.1: Descriptions algorithmique et architecturale.

6.4.4 Codage matériel

L'étape de validation est aussi la spécification pour le codage VHDL. Puisque nous avons construit le prototype virtuel (description architecturale) de manière à ce qu'il soit aussi proche que possible de l'architecture cible, l'étape de codage se réduit à une simple translation du code en langage C. Le concepteur se concentre uniquement sur la syntaxe VHDL ainsi que sur l'optimisation de la description VHDL sans se soucier de la fonctionnalité déjà validée.

La validation et le codage étant découplés, une erreur dans cette étape ne peut provenir que du code VHDL. Le concepteur dispose maintenant d'un moyen puissant pour la localiser: il lance deux exécutions C/VHDL en pas à pas et compare le contenu des points mémoires des deux descriptions. Lorsque le code VHDL est stable, il peut également comparer la réponse des deux descriptions à une série de stimuli.

Nous utilisons le même exemple que précédemment (tableau 6.2) pour illustrer l'obtention de la description matérielle depuis la description architecturale. Les instructions C sont remplacées par la syntaxe VHDL. La translation n'est pas toujours aussi aisée puisque la création de processus peut-être plus délicat à obtenir.

description architecturale	description matérielle
synd = calloc(2t, sizeof(int))	P-Syndromes : PROCESS(clk)
for(all-cycles clk) {	IF clk'EVENT AND clk = '1' THEN
for(all-synd k) {	FOR k IN 2*T-1 DOWNTO 0 LOOP
synd(k) ← proc-recursive(synd(k))	synd(k) <= proc-recursive(synd(k), α^k)
}	END LOOP;
}	END PROCESS P-Syndromes;

Tableau 6.2: Descriptions architecturale et matérielle.

6.4.5 Les étapes de conception

La figure 6.7 illustre notre approche de conception: entre spécification et description matérielle, nous ajoutons l'étape d'obtention d'un couple algorithme-architecture et de l'écriture d'un modèle de validation.

Comme le modèle en cascade, il traite séquentiellement les domaines cognitifs, fonctionnels et matériels, puisque le codage VHDL traduit scrupuleusement le modèle de validation. Toutefois, cette sérialisation est plus souple que le modèle initial dans le sens où une étape peut commencer mais jamais finir avant que l'étape précédente soit elle-même finie. Ceci permet de coder une partie de la description matérielle pour annoter le modèle de validation et donc de renseigner le concepteur logiciel sur

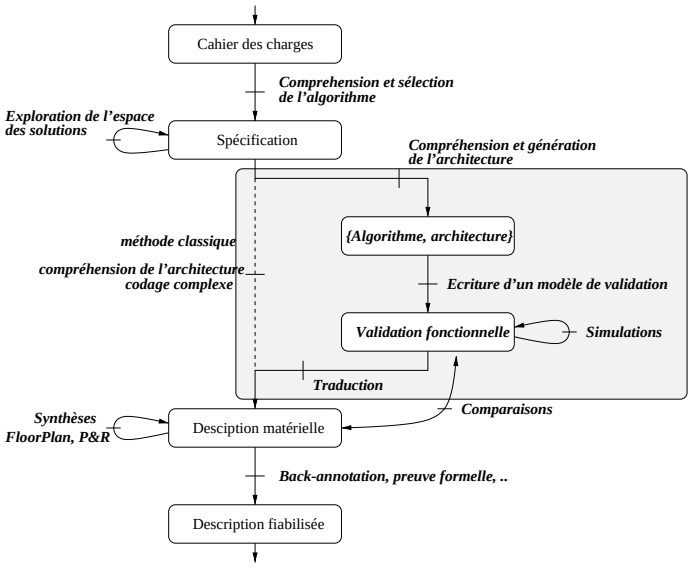

Figure 6.7: Réalisations cognitive, fonctionnelle et matérielle.

la faisabilité et le coût d'une architecture.

Comme le modèle en spirale, il traite parallèlement le domaine matériel, temporel et physique. Le but est de comprendre l'impact d'une description VHDL sur la surface et le chemin critique, ainsi que sur le placement et le degré de congestion. Au vu de la spécification, le concepteur sait si sa description remplit le cahier des charges ou doit être encore optimisée. La séparation du placement et routage de la synthèse conduit souvent à une congestion trop importante, voire à un échec de routage.

Comme le modèle par correction, il permet au concepteur de traiter les problèmes difficiles l'un après l'autre. A partir d'une description naïve, on ajoute successivement: (1) l'**optimisation** pour fournir un circuit plus petit, rapide et facilement synthétisable, (2) la **généricité** pour rendre le code paramétrable, et (3) la **documentation** pour le rendre facilement sélectionnable et intégrable. Ceci permet, comme précédemment, de localiser une erreur de codage.

6.5 Flot d'intégration d'une IP

Cette section décrit l'ensemble des étapes à effectuer pour intégrer une IP. Il s'agit de ce que le concepteur doit s'attendre à trouver lors de l'acquisition d'un composant virtuel de correction d'erreur. La section suit fidèlement les critères d'efficacité d'une méthode de réutilisation.

6.5.1 La sélection

Une bonne documentation est essentielle pour déterminer l'adéquation d'une IP particulière pour une application donnée. La documentation que nous avons mise au point se compose de deux parties: 1) un ensemble de **feuilles de données** (*datasheet*) dont un exemplaire est fourni en annexe pour le code de Reed-Solomon, et 2) un **guide d'utilisateur** qui fournit assez d'information pour comprendre la structure et le mode d'intégration du composant. Les fiches de données permettent de sélectionner rapidement une IP au vu de ses caractéristiques essentielles. Celles-ci décrivent:

- la fonctionnalité du bloc,
- la description des entrées/sorties avec leur fonction,
- le diagramme du bloc,
- les performances pour une cible clairement définie dans des conditions opératoires précises,
- la surface et le nombre de portes équivalentes,
- la structure de testabilité,
- les paramètres et information de configuration,
- la description du protocole de communication,
- des chrono-grammes montrant la communication et le traitement des données,
- la description de l'architecture choisie lorsque celle-ci est référencée dans l'état de l'art.

Les fiches de données fournies en annexe décrivent l'IP de Reed-Solomon dans les trois versions d'architectures proposées: outre l'encodeur unique, nous proposons une architecture massivement parallèle, une architecture micro-contrôlée, et une architecture mixte intermédiaire. Une partie seulement des fiches sont communes tandis que la fonction des ports, le protocole ainsi que les performances sont spécifiques à chaque version.

Une fois l'IP sélectionnée, le guide d'utilisateur permet de comprendre la constitution du bloc réutilisable. Il permet au concepteur d'effectuer la phase de spécialisation et d'intégration afin de réutiliser l'IP. Le guide contient, en plus des fiches techniques précédentes:

- la description de la hiérarchie des sources HDL nécéssaires,
- la structure des répertoires et leur contenu,

- le fonctionnement du démonstrateur logiciel,
- le fonctionnement du programme de personnalisation des sources HDL,
- le fonctionnement du programme de génération des stimuli,
- la stratégie d'optimisation et la structure des scripts de synthèse,
- l'analyse et la compilation des sources HDL,
- la structure et le fonctionnement des *test benches*.

6.5.2 La personnalisation

La personnalisation se fait par l'intermédiaire de programme logiciel au vu des résultats présentés par le démonstrateur. Elle peut s'effectuer selon deux méthodes: d'abord une **paramétrisation** effectue une spécialisation de l'IP avant la synthèse. De cette manière, le composant synthétisé est spécifique et donc de surface minimale. D'un autre coté, on peut aussi synthétiser une IP générique et la **configurer** dans une phase d'initilisation avant chaque utilisation. L'IP est plus polyvalente mais plus grosse car elle doit supporter le pire cas de chaque paramètre. La figure 6.8 résume les deux approches: les composants spécifiques possèdent les mêmes fonctionnalités mais celui issu de la spécialisation avant synthèse est plus petit. L'option choisie par Thomson TCS est la paramétrisation puisque c'est la solution conduisant à la surface minimale, ce qui représente la contrainte primordiale.

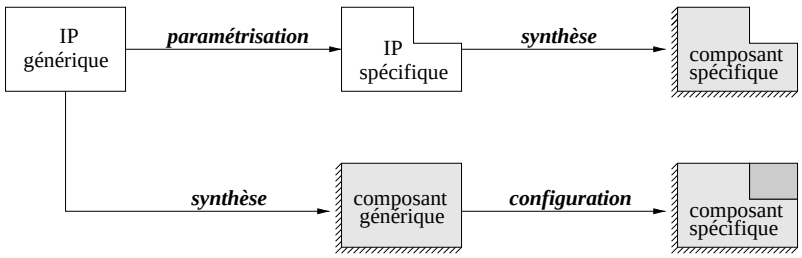

Figure 6.8: Paramétrisation et configuration.

Les sources HDL sont composées de paquetages et de couples entités/architectures. Les paquetages génériques rassemblent des fonctions et constantes uniquement: les constantes définissant la structure de chacun des codes: longueur, capacité, polynômes générateur, etc... , et des constantes d'élaboration permettant de fournir directement à l'outil de synthèse des constantes pré-définies au lieu d'expressions combinatoires à élaborer. Par exemple, l'inversion d'un élément donné du corps de Galois $\alpha \in GF(2^8)$ s'exprime soit par une constante $\beta = \alpha^{-1}$, soit par l'expression α^{254}, à éviter puisqu'elle nécessite 254 multiplications. La paramétrisation des paquetages

permet d'optimiser la phase de synthèse en soulageant l'outil de l'élaboration de constantes combinatoires complexes.

Le second avantage de la personnalisation par programme logiciel est le suivant: lancer une exécution du prototype virtuel permet de collecter des informations quant à la position et l'ordonnancement des données dans le composant. Prenons l'exemple du décodeur de Viterbi de contrainte 8, constitué de 128 processeurs: il semble à priori obligatoire de tester la sortie des 128 processeurs à chaque cycle afin de déterminer celui qui fournit l'information requise. Toutefois, puisque le treillis est cyclique sur une période de longueur 7, il ne peut y avoir plus de 7 processeurs susceptibles de fournir une information particulière. Le prototype virtuel permet de déterminer expérimentalement cet ensemble de processeurs et donc de diminuer le multiplexage de 128 à 7 entrées ! La figure 6.9 illustre comment l'exécution du prototype virtuel permet de déterminer expérimentalement les processeurs actifs et donc de réduire la taille du multiplexage correspondant. La paramétrisation des paquetages permet donc de diminuer la complexité du code RTL en fonction du comportement du prototype virtuel.

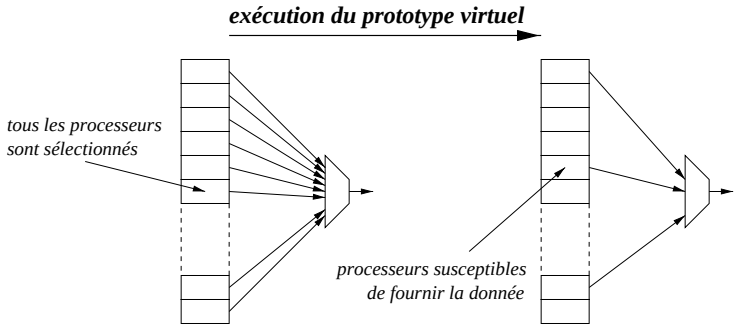

Figure 6.9: Exécution du prototype virtuel.

Le démonstrateur contient le prototype virtuel de l'IP encapsulé dans un programme gérant les paramètres et les contraintes de l'utilisateur. Il permet d'effectuer le codage, la corruption des mots du code et la correction des fichiers de données. Lorsque la dimension du corps de Galois est $m = 8$, il peut effectuer ces procédures sur des fichiers ASCII et présenter visuellement les opérations algébriques. Au vu des résultats, le concepteur peut choisir de réaliser le composant matériel correspondant au prototype. Le programme de personnalisation transforme donc les paquetages génériques en paquetages spécifiques à une application particulière. Le flot de personnalisation est illustré avec le décodeur de Viterbi suivant:

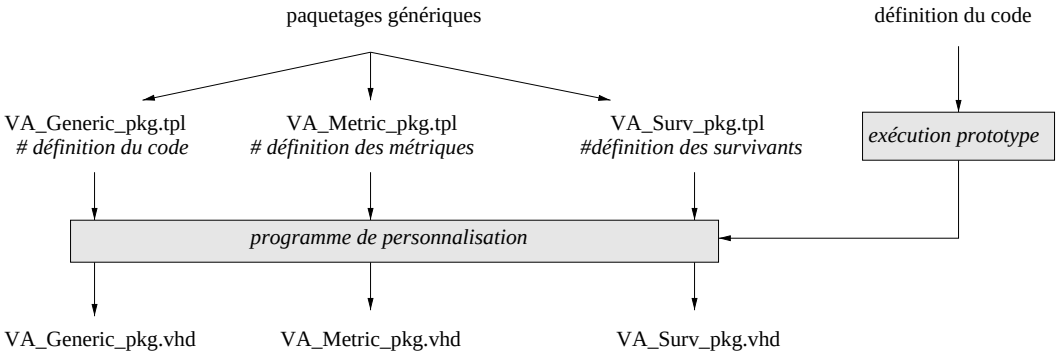

Figure 6.10: Paramétrisation des paquetages.

6.5.3 L'intégration

L'intégration des IPs est permise par l'utilisation d'un style de codage universellement reconnu, ainsi qu'un ensemble de stratégies de synthèse et de scripts. Le style de conception est conforme aux manuels de méthodologie diffusés par les sociétés de micro-électronique. Il s'attache notamment à ce que:

- le système soit totalement synchrone et basé sur l'utilisation de registres uniquement, éliminant totalement les *latches* dont le comportement temporel est ambigu,
- le nombre d'horloge soit minimal (une seule en général) avec des intéractions dans le plus petit module possible,
- un reset asynchrone soit utilisé bien que plus difficile à synthétiser,
- une convention globale de la lexicographie des noms soit utilisée,
- une insertion systématique des en têtes de commentaires de base de données *sccs* soit effectuée, indiquant l'auteur, la date, le nom du fichier et l'historique des modifications apportées,
- une indentation et des commentaires significatifs soient employés,
- l'écriture des entités, architectures et configurations d'un bloc soit stockée dans le même fichier,
- les fonctions et procédures soient préférées aux processus combinatoires répétés,
- des types standards IEEE soient référencés uniquement, en utilisant de préférence des types résolus,

- les constantes et les paramètres soient contenus dans des fichiers indépendants appelés paquetages, plutôt que cablés dans le code source,
- un seul front d'horloge soit utilisé, voire les deux exceptionnellement mais dans deux entités différentes,
- aucun *buffer* ne soit inséré sur les signaux d'horloge ou de reset,
- un code synthétisable seulement soit écrit avec une complète liste de sensibilité et utilisant des signaux plutôt que des variables pour assurer la compatibilité des circuits avant et après synthèse,
- le circuit soit partitionné en sous-blocs de taille synthétisable dont les sorties soient systématiquement mémorisées.

Tout cela assure une portabiblité du code RTL ou comportemental sur l'ensemble des outils de synthèse commerciaux.

Les stratégies de synthèse consistent à développer un ensemble de contraintes pour le composant virtuel le plus tôt possible dans le processus de conception. La stratégie consiste à définir un budget temporel en précisant l'horloge, les temps de *setup* pour les signaux entrants, les temps d'arrivée ou de départ pour les chemins combinatoires traversant le composant, des cellules d'alimentation et de charge pour l'interface du composant, des conditions opératoires (voltage et température), un ou plusieurs modèles d'interconnexion. En ce qui concerne les groupes de fils, il est important de connaître la surface physique du bloc s'il est placé hiérarchiquement, ou la surface globale de l'IP si elle est mise à plat par l'outil de placement routage. Enfin, remarquons que tous les blocs durs (RAM et ROM) doivent être instantiés dans une enveloppe prévue à cet effet, effectuant l'interface entre le code générique et le composant dur figé. Le concepteur n'a donc pas besoin de changer le code source pour modifier l'IP.

Les scripts de synthèse quant à eux réalisent un compromis entre les approches descendantes et ascendantes. En effet, les outils de synthèse classiques effectuent une compilation descendante par défaut, c'est-à-dire que le circuit fourni en entrée est optimisé ainsi que tous les sous-blocs qui y sont contenus. L'avantage de cette approche est une optimisation automatique globale de tous les blocs, mais exige beaucoup de mémoire et de temps de synthèse. Par conséquent, on s'oriente vers une approche ascendante qui consiste à compiler tous les sous-blocs de l'IP les uns après les autres, pour un budget de compilation minimal, puis de les intégrer dans le bloc immédiatement supérieur. L'approche Compiler-Caractériser-Recompiler consiste à:

1. effectuer une compilation de chaque sous-bloc avec les contraintes par défaut à son interface (script *CCC-1.scr*),

2. caractériser le composant dans sa globalité, c'est-à-dire extraire les vraies contraintes aux interfaces, pour prépaper une optimisation temporelle le plus souvent (script *CCC-2.scr*),

3. Recompiler le manière incrémentale chaque sous-bloc (script *CCC-2.scr*).

Le concepteur indique donc les blocs qu'il veut compiler indépendamment et caractériser. Les autres blocs et leur hiérachie sont optimisés avec le bloc supérieur les contenant. L'approche descendante est donc choisie par défaut lorsque le concepteur ne précise pas une approche ascendante. Les scripts, comme les sources HDL, ont leurs propres en têtes *sccs*, contiennent le maximum de commentaires décrivant la stratégie de compilation, ne contiennent aucun nom, valeur ou chemin cablés, mais sont au contraire paramétrables par la liste de blocs à compiler ainsi que leurs contraintes d'optimisation (*CCC-blks.lis*).

6.6 Vérification des IPs

6.6.1 Fichiers de simulation

L'objectif des fichiers de simulation (*test benches*) est la simulation des composants virtuels et la comparaison avec son prototype virtuel. La structure des fichiers HDL de simulation est une hiérarchie d'entités de simulation. L'entité la plus large est le *test bench* et contient à la fois la génération de stimuli et l'observation de la réponse du circuit. Il contient, comme indiqué sur la figure 6.11, cinq entités principales.

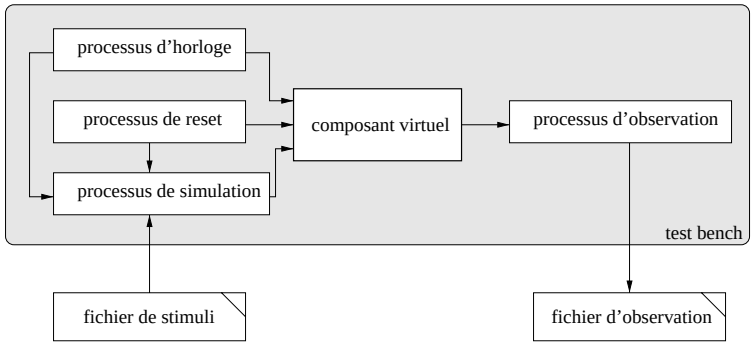

Figure 6.11: Hiérarchie d'entités de simulation.

- le processus d'horloge génère un signal d'horloge qui alimente le composant et synchronise les autres processus,

- le processus de reset, synchrone ou asynchrone, met le composant dans un état connu et teste la réponse du circuit,

- le processus de simulation génère les stimuli et les fournit au circuit en respectant le protocole d'interface. Pour rendre le *test bench* générique et indépendant des vecteurs de simulation choisis, le processus lit dans un fichier externe l'ensemble des vecteurs de test. Il change également le format des données pour les rendre interprétables par le circuit et communique avec l'IP via son protocole,

- le processus d'observation scrute les sorties du circuit pour récupérer les données et les stocke dans un fichier externe correspondant aux stimuli du fichier d'entrée. Il peut également changer le format des données pour les rendre directement lisibles par le concepteur,

- le composant virtuel enfin est l'entrée HDL à simuler. Il s'agit d'instancier un composant en connectant les processus précédents aux ports du circuits.

6.6.2 Génération de stimuli

La validation cherche à obtenir le **défaut zéro** pour toutes les configurations possibles, et cela en un temps minimum. La qualité d'un bloc réutilisable est primordial pour la raison suivante: supposons qu'un circuit soit construit à partir de dix blocs réutilisés dont la probabilité de bon fonctionnement soit de l'ordre de 95%. Alors, la probabilité de bon fonctionnement du circuit devient $(0.9)^{10} = 60\%$. De plus, si chaque bloc a N entrées, il faut 10.2^N tests, s'ils sont effectués individuellement, contre 2^{10N} s'ils sont faits en groupe. En conclusion, nous disons qu'il faut vérifier le plus sûrement possible les blocs élémentaires avant de les assembler, et ceci reste valable pour les sous-blocs d'un bloc. La stratégie de validation consiste donc à valider les sous-blocs de la macro avant de valider le bloc lui-même.

Les types de vérification, obtenus par des outils de simulation, sont les suivants:

- **Test de conformité**: il verifie que le bloc soit conforme avec la spécification.

- **Test en coin**: il simule un ensemble de stimuli inhabituels pouvant mettre le bloc en difficulté puisque le concepteur connaît les points critiques de son IP.

- **Test aléatoire**: il détecte les erreurs que le concepteur ne soupçonnait pas lors des tests spécifiques.

- **Test réel**: il place le bloc dans une application industrielle afin de vérifier que la spécification correspond bien aux attentes des utilisateurs.

Le nombre de stimuli possible est excessivement important pour un décodeur de Reed-Solomon et infini pour un décodeur de Viterbi puisque la séquence en entrée peut être elle-même infinie. Pour le premier codec, en considérant un code de longueur N, de capacité T défini sur un corps de Galois de dimension m, le nombre de vecteurs

possibles est: $2^{N.M}$. Si on ne considère que le motif d'erreur superposé sur un même mot du code, il y a tout de même:

$$C_N^P.(2^M - 1)^P \tag{6.1}$$

vecteurs d'erreur de poids P. Il faut s'assurer que les motifs d'erreur de poids inférieure ou égale à T sont corrigés, ou inférieur ou égal à $2T$ si on considère également les effacements. A titre indicatif, un code (73, 53, 10) en totalise $C_{73}^{10}.2^{80} = 7.5.10^{35}$. On réalise donc les tests suivants:

- test exhaustif: pour les petites instances, on génère tous les vecteurs possible. Par exemple, un code (7, 3, 2) sur $GF(2^3)$ possède $(2^3)^7 = 2^{21} = 2M$ vecteurs possibles pour une simulation exhaustive. Le test exhaustif est très important: puisque le composant virtuel est parfaitement générique, il permet d'atteindre tous les chemins et les situations alors que ceux-ci n'étaient pas testables par un stimulus facilement identifiable.

- test de conformité: on génère un ensemble de mots, corrompus ou non, entrant dans le décodeur de manière consécutive ou non, afin de tester le protocole de communication, l'entrée et la sortie des données ainsi que la fonctionnalité de tous les ports de sorties.

- test en coin: on génère toutes les erreurs de poids 1, soit $N.2^M$, pour vérifier que le décodeur peut atteindre toutes les positions et les amplitudes possibles. Pour chaque poids de l'erreur et pour un ensemble particulier d'amplitude, on applique toutes ou un maximum de positions possibles parmi $C_N^{k\leq T}$ pour tester ainsi la génération du polynôme locateur d'erreur. Pour un motif d'erreur, on applique toutes ou un maximum d'amplitudes possibles: $(2^M-1)^{k\leq T}$ pour tester la génération du polynôme évaluateur d'erreur.

- test aléatoire: on génère des motifs en rafale ou aléatoires de poids inférieur ou égal à T (ou $2T$ avec effacements) de positions et d'amplitudes aléatoires. On génère aussi des vecteurs quelconques dépassant pour certain la capacité du code afin de tester la détection d'échec de décodage.

Le programme de génération de ces stimuli accepte donc en entrée: la définition du code, le type de test choisi, le type d'erreur (aléatoire ou en rafale), le poids maximal d'une erreur ainsi que le nombre de vecteurs désiré.

6.6.3 Méthode de simulation

Le *test bench* dépend du circuit testé dans la mesure ou le format des données (dimension du corps da Galois et polynôme primitif) dépend du code correcteur d'erreur utilisé. La génération du *test bench* est donc réalisée automatiquement par le programme de personnalisation. L'ensemble du flot de simulation est le suivant: on

génération automatiquement des stimuli pour former le fichier en entrée du processus de simulation, on crée le *test bench* associé au code à partir d'un fichier générique en personnalisant les processus de simulation et d'observation, on analyse d'une instance, synthétisée ou non, du code considéré jusqu'à atteindre la fin du fichier de simulation, on simulation du prototype virtuel en langage C du même fichier de simulation, et on comparaison automatique des deux fichiers d'observation et compte rendu des éventuelles divergences.

Le schéma ci-dessous illustre la méthode de simulation totalement automatisée. Elle effectue la simulation et la comparaison d'un composant HDL et de son prototype virtuel.

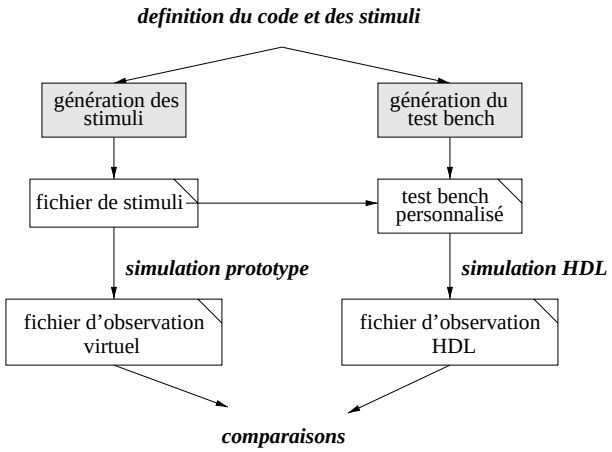

Figure 6.12: Méthode de simulation.

6.6.4 Synthèse et preuve

La simulation au niveau porte a été autrefois la verification finale avant la fabrication d'un ASIC. Pour des circuits de quelques dizaines de milliers de portes, cette simulation n'est simplement pas faisable puisque même avec des accélérateurs matériels, la vitesse n'est pas suffisante. De plus, il est souvent nécessaire de répéter la simulation dans les cas nominaux, les meilleurs, ou les pires. A présent, on effectue:

- une vérification formelle pour vérifier que la *netlist* synthétisée correspond à la description RTL synthétisable,
- une insertion de scan qui complète les tests de fonctionnalité,
- une analyse temporelle après placement et routage pour vérifier que les contraintes temporelles et la fréquence d'horloge sont effectivement tenues.

La preuve formelle utilise des techniques mathématiques pour prouver l'équivalence de deux représentations du circuit. Il est donc inutile d'appliquer des vecteurs de test fonctionnels. Les circuits peuvent être comparés aussi longtemps qu'ils ont la même fonctionnalité synchrone. Ils sont équivalents si la fonction est la même pour les sorties et pour chaque point de mémorisation.

6.6.5 Placement et routage

Le placement et routage constitue à la fois une étape nécessaire dans la conception de l'IP pour une future intégration, mais aussi une validation de l'architecture générée. Les principaux objectifs sont: la minimisation de la surface de silicium, la tenue des contraintes temporelles (fréquence d'horloge notamment), la minimisation des incertitudes temporelles sur l'arbre d'horloge et de reset (*skew*) ainsi que la constitution d'un réseau d'alimentation pour prévenir les chutes de tension et l'électromigration.

L'apport du placement routage consiste à caractériser les performances d'un bloc en tenant compte des contraintes physiques du routage: évitement des congestions et pénalisations temporelles dues aux effets résistifs et capacitifs. Bien que la synthèse puisse sélectionner un modèle d'interconnexion statistique dont les valeurs dépendent de la surface, le problème est qu'un placement routage spécifique ne conduit jamais aux prédictions statistiques. Deux solutions ont été envisagées dans le cas des codes correcteurs d'erreur:

- d'abord on utilise un outil permettant de déterminer la forme et la place des principaux blocs de l'IP (*Design Planner*). Il peut déduire du nombre de portes équivalentes une surface de silicium et surtout un modèle d'interconnexion suffisamment prédictif pour être intégré dans une étape de synthèse, puisqu'il utilise la même approche que l'outil final de placement routage. Cette méthode a été choisie pour le bloc de Reed-Solomon.

- la deuxième solution, sélectionnnée pour le bloc de Viterbi dont les performances temporelles sont critiques, consiste à réaliser une réalisation physique plus poussée qu'un pré-placement mais nécessite un cycle plus long: la synthèse fournit un fichier de contraintes temporelles qui dirige le placement routage des cellules standards de telle sorte que les chemins critiques soient traités en priorité.

Dans une phase ultérieure, il faut construire un arbre d'horloge. Alors que la synthèse ne tient pas compte du placement physique des points de mémorisation, l'outil de routage réalise des regroupements en fonction de leur position de telle sorte que l'arbre soit équilibré en minimisant le délai de propagation et le *skew* d'horloge. Notons enfin qu'une optimisation en place *IPO*) permet de modifier la puissance des portes et d'insérer des *buffers*, et donc d'accroître les performances temporelles d'un

circuit sans en modifer le placement initilal. Il n'est donc pas utile de réitérer tout le cycle de placement routage. On fournit en annexe un schéma de placament e routage de l'IP Reed-Solomon.

6.6.6 Prototypage

Les circuits de petites tailles, c'est-à-dire les codes correcteurs d'erreur de faibles capacités, sont prototypés sur un FPGA. Ils ont l'avantage d'être reprogrammables permettant une rapide reconfiguration pour la correction d'erreurs. Les FPGA sont très appropriés pour les petits circuits d'autant plus qu'ils rejoignent les performances des ASIC en nombre de portes et fréquence d'horloge. L'étape de prototypage est en cours de réalisation dans un projet conjoint entre Thomson TCS et le laboratoire CSI de l'INPG.

6.7 Réflexions sur la conception d'IP

6.7.1 Difficultés rencontrées

La première difficulté rencontrée a consité à évaluer les performances et le coût d'une architecture avant d'effectuer le codage HDL. Il fallait également s'assurer que l'architecture proposée par l'état de l'art ou que nous avons nous-mêmes construite était fonctionnellement correcte. Faute de quoi, nous aurions dû réitérer l'ensemble du processus de conception, alors que nous n'avions certainement pas le temps pour le faire. L'ensemble de ces incertitudes nous a conduit à introduire la notion de prototype virtuel et à concevoir en langage C les architectures des IPs à réaliser en langage matériel. L'inconvénient important de cette approche est que si la fonctionnalité est très bien testée ainsi que quelques éléments de performance (latence et débit par exemple), elle ne permet pas d'évaluer précisément la surface et la fréquence d'horloge qui sont fortement liées à la réalisation matérielle. Ceci peut amener à choisir une architecture alors que celle-ci peut s'avérer trop grosse. Prenons l'exemple du décodeur de Reed-Solomon: afin de couvrir toutes les exigences de débit, nous avons mis au point quatre architectures: deux extrêmes (1) parallèle et 2) micro-contrôlée) et deux mixtes (3) une version parallèle dans laquelle on introduit un traitement séquentiel des opérations arithmétiques dans le corps de Galois et 2) une version micro-contrôlée dans laquelle les coefficients d'un même polynôme sont traitrés simultanément). Ces quatre versions ont été successivement prototypées, validées puis codées en VHDL RTL. Malheureusement, la quatrième d'entre elles a été abandonnée pour la simple raison que le traitement simultané des coefficients des polynômes nécessitait le stockage de tous les coefficients dans des mémoires distinctes. Par conséquent, la RAM de stockage est divisée en plusieurs RAM plus petites ce qui provoque une augmentation importante de la surface. Il était alors plus rentable d'utiliser la troisième solution avec une contrainte de surface sévère pour obtenir un débit plus important pour une

surface identique. Après avoir réalisé ces architectures, il serait possible d'annoter le code C pour estimer la surface d'une architecture. En effet, une fois connue la surface des points mémoire, additionneurs et multiplieurs dans le corps de Galois, et des blocs durs, il devient possible de comptabiliser le nombre d'opérations et de d'unités de stockage pour fournir une estimation réaliste de la surface active.

La seconde difficulté rencontrée consiste à effectuer les vérifications physiques en fin de conception (*design front-end*). Bien que nous ayons précédemment indiqué que la surface active était un bon estimateur de la surface finale du circuit, il est également vrai que le style de codage et l'architecture peut influencer significativement le placement-routage. C'est en effet le cas pour l'architecture TPI paramétrable du décodeur de Viterbi qui se compose pour l'essentiel de RAM de stockage, de processeurs combinatoires et de multiplexage. Le routage des arbres de multiplexeurs provoque une augmentation importante de la surface. C'est pourquoi nous avons utilisé de préférence des *buffers* troisième état nécessitant une connexion de moins et donc un meilleur comportement au routage. Il n'est pas possible pour l'instant de prendre en compte les difficultés de la réalisation physique dans la phase de conception. Le cas s'est également produit dans les blocs arithmétiques de la technologie Xilinx. Une architecture de multiplieur basée sur une décomposition dichotomique en additionneurs s'est révélée inroutée pour les grandes opérandes, dûes à l'augmentation très importante du nombre de connexions entre additionneurs successifs. L'étape de validation nous a forcé d'éliminer totalement cette architecture.

La troisième difficulté rencontrée consiste à vérifier le bloc réutilisable. Ceci est réalisé complètement pour les blocs arithmétiques de petites tailles tels que les additionneurs de moins de 32 bits et les multiplieurs de moins de 16 bits. En effet, en considérant un autre bloc de même fonctionnalité que l'on sait fonctionnellement correcte (ceci est facilement accessible par l'utilisation d'architectures triviales), on représente les deux blocs sous forme d'un arbre de décision binaire (BDD) muni d'un même ordre de variable, afin de rendre la représentation canonique. Par simple comparaison structurelle, on assure un parfaite correspondance des deux architectures. En ce qui concerne les codes correcteurs d'erreur, l'introduction d'éléments séquentiels ne permet pas de vérification booléenne. De plus le nombre de vecteurs de test est extrêmement important. On réalise donc: 1) une vérification exhaustive de petites architectures, 2) un ensemble important de tests de conformité, en coin et aléatoires, 3) de la preuve formelle pour valider l'étape de synthèse, et 4) le prototypage d'un composant spécifique. L'avantage de la dernière étape réside dans un délai de simulation rapide et en environnement réel.

6.7.2 Vers les blocs durs

Les IPs dures sont les blocs délivrés avec une représentation physique au format GDSII. L'avantage est évident: les performances en termes de surface, de puissance dissipée et de fréquence d'horloge sont parfaitement déterminées. Toutefois, elles perdent leur flexibilité et leur portabilité puisqu'une resynthèse se traduit maintenant par une étape de migration technologique.

Concevoir une IP dure n'est pas différent de concevoir une IP souple (*soft macros*) mais il faut en plus générer une représentation physique et développer des modèles pour la simulation et le *layout*. Les IPs dures doivent aussi intégrer une stratégie de testabilité: soit une chaîne de test (*scan*) qui est peu pénalisante en surface et en délai mais qui nécessite une mémoire de test externe, soit un *BIST* logique qui, malgré une augmentation significative de la surface empêche tout étude dite de *reverse engineering* puisqu'il génère ses propres vecteurs de test. Signalons en plus que l'IP doit devancer les contraintes du circuit global: l'arbre d'horloge doit être construit sans connaître la structure d'horloge du circuit englobant, avec un minimum de charge, la forme et le placement des pines de la macro doivent permettre une intégration aisée, les pistes d'alimentation enfin doivent supporter les pics de courant maximum. Ainsi, la sélection d'une IP dure ne se fait pas seulement sur la documentation ou l'environnement de vérification mais aussi sur la robustesse du circuit qui doit avoir été vérifié sur silicium ainsi que les limitations physiques précédemment présentées.

6.7.3 Avantages des IPs souples

Les IPs souples telles que nous les avons réalisées présentent au moins trois avantages comparés aux macro-cellules ou blocs "non" réutilisables: elles sont d'abord parfaitement génériques et spécialisables en fonction des contraintes du concepteur. Des décodeurs de Reed-Solomon existent mais sont en général fournis dans une seule configuration. Or, les applications que nus avons rencontrées sont dominées par le coût de silicium, et donc nécessitent la plus petite surface dans des contraintes minimales de débit. Sachant que les exigences de débit peuvent varier entre une application spatiale de faible débit et une application de video-surveillance de fort débit, il est nécessaire de posséder une palette d'architectures de même fonctionnalité. Aussi les décodeurs sont fournis en plusieurs rapports surface/débit, ce qui fait la différence avec les blocs classiques "non" réutilisables.

Elles sont totalement indépendantes du processus technologique et portables sur toutes les cibles technologiques. En effet, les sources HDL sont écrites au niveau transfert de registres synthétisable, sans aucune référence à la bibliothèque cible. Il faut prendre soin de définir des contraintes de synthèse réalistes et une première compilation permet de connaître les possibilités d'une nouvelle cible. Nous avons synthétisé nos IPs sur trois cibles différentes: une technologie 0.8 μm de Thomson

TCS, une technologie 0.35 μm de ST Microelectronics, et une bibliothèque *Altera* de FPGA afin d'effectuer le prototypage. Ceci en fait une différence importante avec les macro-cellules pour lesquelles l'assignation technologique est une étape importante. Les évolutions technologiques, l'augmentation du nombre de couches de métal et la diminution des géométries sont prises en compte par le processus classique de synthèse et ne modifie pas la description de l'IP. Le niveau d'abstraction élevé est une caractéristique importante.

Enfin, elles sont intégrables dans n'importe quelle application plus importante dont une des descriptions est commune avec un des niveaux d'abstraction de l'IP, comme par exemple du VHDL RTL ou un réseau de portes. Les scripts de synthèse permettent d'intégrer le composant selon les objectifs et les contraintes du concepteur. L'optimisation peut s'arréter dans un format interne à l'outil de synthèse Synopsys, au niveau réseau de portes VHDL, Verilog ou EDIF. Cette propriété de portabilité dans un processus de synthèse plus important rend le bloc particulièrement facile à réutiliser.

6.8 Elever le niveau d'abstraction

Nous proposons ici un résumé des comparaisons entre les deux niveaux comportemental et RTL pour l'ensemble des deux blocs de codage réalisés: Reed-Solomon et Viterbi. Les trois critères quantitatifs que nous pouvons comparer sont: le temps de conception, la longueur de code et la qualité de résultat *QoR*. Alors que le premier critère n'intéresse que le concepteur, le réutilisateur se préoccuperra des deux autres puisqu'ils conditionnent la facilité et la qualité de l'intégration.

- On constate un rapport de **temps de conception** (codage matériel uniquement sans prendre en compte la compréhension et la validation) du RTL par rapport au comportemental d'un facteur 3 ou 4: 3/11 semaines pour le Viterbi, 6/24 semaines pour le Reed-Solomon. Bien que les principales fonctions utilisées dans le comportemental le sont aussi dans le RTL, les détails d'architecture du code RTL sont réalisés automatiquement par le comportemental en fonction des contraintes de l'utilisateur.

- La **QoR** du RTL est en général meilleure pour deux raisons: pour une architecture identique, le circuit est souvent plus petit. Il est parfois aussi plus rapide en évitant des communications répétées avec la partie contrôle. Dans tous les cas, la hiérarchie de code RTL permet une compilation plus poussée. Mais la seconde raison est la plus importante: toutes les architectures RTL ne sont pas atteignables par un code comportemental, comme le montre l'exemple du décodeur de Viterbi. Outre l'ordonnancement des opérateurs, nous avons construit une architecture RTL utilisant un ordonnacement particulier des opérandes

conduisant à un gain de débit pour une même surface. Le comportemental a lui un débit proportionnel au nombre de ressources utilisées.

- Le **nombre de lignes de code** est 10 fois plus grand pour le RTL: 307/2943 pour le Viterbi, 581/11393 pour le Reed-Solomon. Notons que plusieurs codes RTL sont développés pour couvrir plusieurs architectures (parallèle, micro-contrôlée, ...) alors qu'un seul code comportemental est réalisé.

C'est pourquoi nous avons utilisé le niveau comportemental pour parcourir l'espace des solutions: le temps et le volume de conception est très réduit pour une qualité de résultat plus faible que le niveau RTL. Une fois l'architecture retenue, nous la condons en RTL pour une QoR optimale.

6.9 Conclusion

Dans ce chapitre, nous avons présenté les méthodologies de réutilisation et les critères d'efficacités qualitatifs et quantitatifs de ces approches. Lorsque la bibliothèque ne propose pas de blocs à réutiliser, il faut les concevoir. Aussi avons nous présenté les méthodes classiques de conception et proposé une amélioration pour les rendre à la fois simple et sûr. Celle-ci nous a servi à concevoir les blocs de codage. Nous avons également présenté l'ensemble des données délivrées afin d'industrialiser ces composants, ainsi qu'une réflexion sur la conception de blocs mous et durs.

Chapitre 7

Conclusion

Cette étude a porté essentiellement sur la conception des blocs réutilisables à utiliser dans plusieurs niveaux d'abstraction, dans des outils automatiques de synthèse et comme composants virtuels. La clé d'un bloc réutilisable résidant dans sa capacité à être par la suite utilisé, nous les avons rendus aussi génériques et paramétrables que possible. Pour chaque niveau de description, nous avons déterminé les contraintes de conception qui sont donc devenus les paramètres des macro-blocs réalisés. Nous avons par exemple étudié le débit et la surface des blocs de haut-niveau, l'environnement temporel et technologique des blocs arithmétiques et la puissance dissipée dans les outils de décomposition technologique.

L'ensemble du travail réalisé se présente sous deux formes: la conception des blocs arithmétiques et la décomposition technologique sur cellules standards sont incorporés dans un outil de synthèse RTL et logique. A l'issue de l'élaboration d'un code VHDL, nous fournissons les équations optimisées des blocs arithmétiques et nous recouvrons toutes les parties combinatoires du circuit. Par contre, l'étude des blocs comportementaux de codage correcteur d'erreur, effectuée au sein de la société *Thomson-CSF Semiconducteurs spécifiques*, nous a permis de construire un ensemble de composants virtuels dont l'objectif est double: d'abord proposer ces composants comme IP (*Intellectual Property*) aux concepteurs de circuits intégrés désireux d'utiliser de telles fonctionnalités. Deuxièmement, faire partie de circuits ASICs industriels de Thomson-CSF: les validations apportées (simulations, placement et routage, puis prototypage) ont conduit à leur utilisation dans une application de vidéo-surveillance notamment.

Les avantages des blocs réutilisables sont multiples: généricité et flexibilité, indépendance vis-à-vis du processus technologique, intégrabilité dans un circuit intégré. On peut toutefois leur reprocher leur manque de détermination physique comparé aux blocs durs, macro plus robuste dont les performances sont caractérisées avant leur intégration, et vers lesquels il semble important de s'orienter.

L'ensemble de ce travail est inspiré par le souci de pragmatisme, notamment du

à l'intégration industrielle des IPs de codage corecteur d'erreur. Les exigences de productivité rendent la conception de blocs réutilisables impérative et nous pensons en avoir montré quelques exemples d'application.

Chapitre 8

Références

[1] A.A.Jerraya, H.Ding, P.Kission, M.Rahmouni, *Behavioral Synthesis and Component Reuse with VHDL*, Boston, MA, Kluwer-Academic, 1997.

[2] C.W.Krueger, *Software reuse*, ACM Computing Survey, vol. 24, No. 2, June 1992, pp. 131-183.

[3] D.L.Ku, G.de Micheli, *Relative Scheduling under Timing Constraints*, IEEE Trans on CAD, May 1992.

[4] D.Gajshi, N.Dutt, A.Wu, Y.Lin, *High-Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers, Boston, Massachussets, 1992.

[5] R.K.Brayton, G.Hachtel, C.McMullen, A.Sangiovanni-Vincentelli, *Logic Minimisation Algorithms for VLSI Synthesis*, Boston, MA, Kluwer-Academic, 1984.

[6] R.K.Brayton, R.Rudell, A.Sangiovanni-Vincentelli, A.Wang, *MIS: A Multiple-Level Logic Optimization System*, IEEE Transactions on Computer-Aided Design, Nov 1988.

[7] W.B.Frakes, C.J.Fox, *Sixteen questions about software reuse*, Communications of the ACM, 38(6):75-87, June 1995.

[8] N.Weste, K.Eshraghian, *Principles of CMOS VLSI Design, A System Perspective*, Addison-Wesley, Reading, MA, 1985.

[9] S.Turgis, N.Azemard, D.Auvergne, *Explicit Evaluation of Short Circuit Power Dissipation for CMOS Logic Structures*, In Proceedings of the IEEE International Symposium on Low Power Design, pp. 129-134.

[10] L.Benini, M.Favalli, B.Ricco, *Analysis of Hazard Contribution to Power Dissi-*

pation in CMOS IC's, In Proceedings of the 1994 ACM/IEEE International Workshop on Low Power Design, pp. 27-32.

[11] M.Pedram, *Power Minimization in IC Design: Principles and Applications*, In Proceedings of the ACM Transactions on Design Automation of Electronic Systems, Vol. 1, No. 1, January 1996, pp. 3-56.

[12] A.Tyagi, *Hercules: A Power Analyzer of MOS VLSI Circuits*, In Proceedings of the IEEE International Conference on Computer Aided Design, 1987, pp. 530-533.

[13] R.Burch, F.Najm, P.Yang, T.Trick, *A Monte Carlo Approach for Low Power Estimation*, In the IEEE Transactions on VLSI Systems, March 1993, pp. 63-71.

[14] R.Bryant, *Graph-Based Algorithms for Boolean Function Manipulation*, In the IEEE Transactions on Computers, August 1986, pp. 677-691.

[15] F.Najm, *Transition Density, a Stochastic Measure of Activity in Digital Circuits*, In Proceedings of the Design Automation Conference, 1998, pp. 294-299.

[16] B.Kapoor, *Improving the Accuracy of Circuit Activity Measurement*, In Proceedings of the 1994 International Workshop on Low Power Design, pp. 111-116.

[17] A.Ghosh, S.Devadas, K.Keutzer, J.White, *Estimation of Average Switching Activity in Combinational and Sequential Circuits*, In Proceedings of the 29^{th} Design Automation Conference, 1992, pp. 253-259.

[18] E.Detjens, G.Gannot, R.Rudell, A.Sangiovanni-Vincentelli, A.Wang *Technology Mapping in MIS*, In Proceedings of the International Conference on Computer Aided Design, 1987.

[19] K.Keutzer, *Dagon: Technology Binding and Local Optimization*, In Proceedings of the Design Automation Conference, 1987.

[20] J.Darringer, D.Brand, W.Joyner, L.Treivillyan, *LSS: A System for Production Logic Synthesis*, IBM Journal of Research and Development, September 1984, pp. 537-545.

[21] D.Gregory et al., *Socrates: A System for Automatically Synthesizing and Optimizing Combinational Logic*, In Proceedings of the Design Automation Conference, 1986, pp. 79-85.

[22] A.V.Aho, S.C.Johnson, *Optimal Code Generation for Expression Trees*, In

Journal of the Assoc.Computer, March 1976, pp. 488-501.

[23] Brace, R.Rudell, R.Bryant, *Efficient Implementation of a BDD Package*, In Proceedings of Design Automation Conference, june 1993, pp. 40-45.

[24] J.R.Burch, D.E.Long, *Efficient Boolean-Function Matching*, In Proceedings of International Conference on Computer Aided Design, november 1992, pp. 90-97.

[25] Sanko H.Lan, *AFG - An Automatic Function Generation For Macro-Cell*, Reasearch Series of Algorithms and Architectures, September 1989.

[26] B.Laurent, *Isomorphismes Booléens sur Librairie*, DEA de Microélectronique, INPG, Septembre 1995.

[27] A.Fortas, B.Laurent, S.A.Senouci, *Methods and Practical Results on Boolean Matching Techniques*, In Proceedings of International Workshop on Logic and Architecture Synthesis, December 1995.

[28] H.Touati, *Performance Oriented Technology Mapping*, Ph.D Thesis, University of California, Berkeley, 1990.

[29] B.Laurent, G.Saucier, *A Tree Driven Matching for Complex Gates Targeting Low Power*, In Proceedings of the 6^{th} International Workshop on Power and Timing Modelling, Optimization and Simulation, September 1996, pp. xx-yy.

[30] M.Crastes and al., *ASYL : a Logic and Architecture DA System*, In Proceedings of Euro ASIC89, Paris, France, January 1989, pp. 183-209.

[31] H.Mehta, M.Borah, R.M.Owens, M.J.Irwin, *Accurate Estimation of Combinational Circuit Activity*, In Proceedings of the Design Automation Conference, 1995, pp. 618-622.

[32] B.Laurent, G.Saucier, *A Low Power Mapping Optimizing Global Power*, In Proceedings of the International Workshop SASIMI, December 1996, pp. xx-yy.

[33] P.M.Kogge, H,S,Stone, *A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations*, IEEE Transactions on Computers, may 1987, pp. 49-56.

[34] J.Sklanski, *Conditional-Sum Addition Logic*, IRE Transactions on Electronic Computers, june 1960, pp. 226-231.

[35] R.T.Brent, H.Kung, *A Regular Layout for Parallel Adders*, IEEE Transactions on Computers, march 1982, pp. 260-264.

[36] T.Han, D.A.Carlson, *Fast Area-Efficient VLSI Adders*, In Proceedings of the 11^{th} Design of Integrated Circuits and Systems Conference (DCIS'96), nov. 1996.

[37] L.Montalvo, K.K.Parhi, *Estimation of Average Energy Consumption of Ripple-Carry Adder Based on Average Length Carry Chains*, IEEE Transactions on Computers, march 1982, pp. 260-264.

[38] S.J.Abou-Samra, A.Guyot, B.Laurent *Spurious Transitions in Adder Circuits: Analytical Modelling and Simulations*, In Proceedings of the Symposium on Computer Arithmetic. 1997.

[39] A.Guyot, M.Belrhiti, G.Bosco *Adder Synthesis*, Logic and Architecture Synthesis, ed. Chapman & Hall, 1994.

[40] G.Bosco, M.Belrhiti, B.Laurent, *Adder Solution Space Exploration for Speed/Area Trade-offs*, In Proceedings of International Workshop on Logic and Architecture Synthesis, December 1996.

[41] B.Laurent, G.Saucier, *Adder Synthesis in Timing Discrepancies*, In Proceedings of International Workshop SASIMI, December 1997.

[42] *The Programmable Logic Data Book, Xilinx 1996.*

[43] B.Laurent, G.Bosco, G.Saucier, *Structural versus Algorithmic Approaches for Efficient Adders on Xilinx 5200 FPGA*, In Proceedings of International Workshop on Field Programmable Logic, September 1997.

[43*b*] B.Laurent, G.Bosco, G.Saucier, *Efficient Arithmetic on Xilinx 5200 FPGA* In Proceedings of International Conference on VLSI Design, September 1997.

[44] S.Oberman, M.Flynn, *Design Issues in Floating Point Division*, Technical Report CSL-TR-94-647, Stanford University.

[45] I.Koren, *Computer Arithmetic Algorithms*, Prentice Hall, 1993, ISBN 0-13-151952-2.

[46] A.D.Booth, *A Signed Binary Multiplication Technique*, Quaterly Journal of Mechanics and Applied Mathematics, 4(2):236-240, Juin 1951.

[47] O.L.McSorley, *High Speed Arithmetic in Binary Computers*, in Proceedings of the IRE, 49(1), January 1961, pp. 67-91.

[48] G.Bewick, M.Flynn, *Binary Multiplication using Partially Redundant Multiples*, Technical Report CSL-TR-92-528, Stanford University.

[49] J.L.Hennessy, D.A.Patterson, *Computer Architecture, a Quantitative Approach*, Morgan-Kaufmann, 1990, pp. A44-A48.

[50] D.Zuras, W.McAllister, *Balanced Delay Trees and Combinatorial Division in VLSI*, IEEE Transactions on Solid-State Circuits, October 1986, pp. 814-819.

[51] Z.Mou, F.Jutand, *A Class of Close to Optimum Adder Trees allowing Regular and Compact Layout*, IEEE Transactions on Computers, 1990, pp. 251-254.

[52] L.Dadda, *On Parallel Digital Multipliers*, Alta Frequenza 45 1976, pp. 574-580.

[53] E.L.Braun, *Digital Computer Design*, New York Academic, 1963.

[54] S.A.Vanstone, P.C.van Oorschot, *An Introduction to Error Correcting Codes with Applications*, Kluwer Academic Publishers, ISBN 0-7923-9017-2.

[55] D.G.Hoffman et al., *Coding Theory - The Essentials*, Marcel Dekker, ISBN 0-8247-8611-4.

[56] R.E.Blahut, *Theory and Practice of Error Control Codes*, Addisson Wesley Publishing Company.

[57] G.D.Forney, *Convolutional Codes I: Algebraic Structure*, IEEE Transactions on Information Theory, November 1970.

[58] A.J.Viterbi, *Convolutional Codes and Their Performance in Communication Systems*, IEEE Transactions on Communications Technology, October 1971.

[59] A.J.Viterbi, *Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm*, IEEE Transactions on Information Theory, April 1967.

[60] J.K.Omura, *On the Viterbi Algorithm*, IEEE Transactions on Information Theory, January 1969.

[61] C.M.Rader, *Memory Management in a Viterbi Decoder*, IEEE Transactions

on Communications, Sept 1981.

[62] S.Bitterlich, H.Meyr, *Efficient Scalable Architectures for Viterbi Decoders*, In Proceedings of the ASAP Conference, 1993.

[63] P.G.Gulak, T.Kailath, *Locally Connected VLSI Architectures for the Viterbi Algorithm,*, IEEE Journal on Selected Areas in Communications, April 1988, pp. 527-537.

[64] G.Feygin, P.G.Gulak, P.Chow, *Generalized Cascade Viterbi Decoder - a Locally Connected Multiprocessor with Linear Speed-up*, In Proceedings of the ICASSP Conference, 1991, pp. 1097-1100.

[65] H.Dawid, S.Bitterlich, H.Meyr, *Trellis Pipeline-Interleaving: a Novel Method for Efficient Viterbi-Decoder Implementation*, In Proceedings of the ISCAS Conference, 1992, pp. 1875-1878.

[66] W.W.Peterson, *Encoding and Error-Correction Procedures for the Bose-Chaudhuri Codes*, IEEE Transactions on Information Theory, pp. 459-470, 1960.

[67] J.E.Meggitt, *Error Correcting Codes and Their Implementation for Data Transmission Systems*, IEEE Transactions on Information Theory, pp. 234-244, 1961.

[68] Y.Sugiyama, M.Kasahara, S.Hirasawa, T.Namekawa, *A Method for Solving Key Equation for Decoding Goppa Codes*, Information and Control, vol. 27, pp. 87-99, 1975.

[69] H.M.Shao, I.S.Reed, *On the VLSI Design of a Pipeline Reed-Solomon Decoder Using Systolic Arrays*, IEEE Transactions on Information Theory, pp. 1273-1280, 1988.

[70] H.M.Shao, T.K.Truong, L.J.Deutsch, J.H.Yuen, I.S.Reed, *A VLSI Design of a Pipeline Reed-Solomon Decoder*, IEEE Transactions on Information Theory, pp. 393-403, 1985.

[71] R.P.Brent, H.T.Kung, *Why Systolic Arrays ?*, IEEE Transactions on Information Theory, pp. xx-xx, xxxx.

[72] R.P.Brent, H.T.Kung, *Systolic VLSI Arrays for Polynomial GCD Computation*, IEEE Transactions on Information Theory, pp. 731-736, 1884.

[73] J.L.Massey, *Shift-Register Synthesis and BCH Decoding*, IEEE Transactions

on Information Theory, pp. 122-127, 1969.

[74] G.D.Forney, *On Decoding BCH Codes*, IEEE Transactions on Information Theory, pp. 549-557, October, 1965.

[75] R.T.Chien, *Cyclic Decoding Procedures for Bose-Chaudhuri-Hocquenghem Codes*, IEEE Transactions on Information Theory, pp. 357-363, October, 1964.

[76] D.V.Sarwate, R.D.Morrison, *Decoder Malfunction in BCH Decoders*, IEEE Transactions on Information Theory, vol. 36, pp. 884-889, July, 1990.

[77] T.Bartee, D.I.Schneider, *An Electronic Decoder for Bose-Chaudhuri-Hocquenghem Error-Correcting Codes*, IRE Transactions on Information Theory, vol. IT-8, pp. S17-S24, September, 1962.

[78] C.S.Yen, I.S.Reed, T.K.Truong, *Systolic Multipliers for Finite Fields* $GF(2^m)$, IEEE Transactions on Information Theory, pp. 357-360, 1984.

[79] B.B.Zhou, *A New Bit-Serial Systolic Multiplier Over* $GF(2^m)$, IEEE Transactions on Information Theory, pp. 749-751, 1988.

[80] C.C.Wang, T.K.Truong, H.M.Shao, L.J.Deutsch, J.K.Omura, I.S.Reed, *VLSI Architectures for Computing Multiplications and Inverses in* $GF(2^m)$, IEEE Transactions on Information Theory, pp. 709-717, 1985.

[81] S.K.Jain, K.K.Parhi, *A Low Latency Standard Basis* $GF(2^m)$ *Multiplier*, Not referenced.

[82] G.Feng, *A VLSI Architecture for Fast Inversion in* $GF(2^m)$, IEEE Transactions on Information Theory, pp. 1383-1386, 1989.

[83] H.Brunner, A.Curiger, M.Hofstetter, *On Computing Multiplicative Inverses in* $GF(2^m)$, IEEE Transactions on Information Theory, pp. 1010-1015, 1993.

[84] C.Wang, J.Lin, *A Systolic Architecture for Computing Inverses and Divisions in Finite Fields* $GF(2^m)$, IEEE Transactions on Information Theory, pp. 1141-1146, 1993.

[85] C.A. Mead, L.A. Conway, *Introduction to VLSI Systems*, Addison-Wesley Publishing Company, 1980.

[86] S.Trimberger, J.A.Rowson, C.R.Lang, J.P.Gray, *A Structured Design Method-*

ology and Associated Software Tools, IECS, 28(7), July 1981.

[87] M.Keating, P.Bricaud, *Reuse Methodology manual for system-on-chip designs*, Kluwer Academic Publishers.

[88] A.Reis, M.Robert, D.Auvergne, R.Reis, *Associating CMOS Transistors with BDD Arcs for Technology Mapping*, Electronic Letters, vol.31, n.14, July 1995.

[89] F.Moraes, L.Torres, M.Robert, D.Auvergne, *Estimation of Layout Densities for CMOS Digital Circuits*, VLSI" Integrated Systems on Silicon, 1997.

[90] F.Moraes, N.Azemard, M.Robert, D.Auvergne, *Flexible Macrocell Layout Generator*, 4^{th} ACM/SIGDA Physical Design Workshop, pp. 105-116, 1993.

[91] C.Chu, M.Pontkonjak, M.Thaler, J.Rabaey, *HYPER: an Interactive Synthesis Environment for High Performance Real Time Applications*, Intl Conference ICCD, pp, 432-435, 1989.

[92] J.S.Lis, D.D.Gajski, *Synthesis from VHDL*, Intl Conference ICCAD, pp. 378-381, 1988.

[93] P.G.Paulin, J.P.Knigh, E.F.Girczyc, *HAL: A Multiparadigm Approach to Automatic Data Path Synthesis*, Intl Conference DAC, 1986.

[94] B.M.Pangrle, *SPLICER: A Heuristic Approach to Connectivity Binding*, Intl Conference DAC, 1988.

[95] Z.Peng, *Synthesis of VLSI Systems with the CAMAD Design Aid*, Intl Conference DAC, 1986.

[96] G.De Micheli, D.C.Ku, *HERCULES - a System for High-Level Synthesis*, Intl Conference DAC, 1988.

[97] Synopsys Inc., *Synopsys Behavioral Compiler User Guide, v. 2.2, 1994.*

[98] A.Mignotte, *Synthèse Architecturale de Circuits Intégrés*, Thèse de doctorat, INPG, 1992.

[99] A.A.Jerraya, I.Park, K.O'Brien., *AMICAL: An Interactive High-Level Synthesis Environment*, European Conference on Design Automation, 1993.

Chapitre 9

Annexe - Fiches Techniques

ISBN 2 - 913329 - 24 - 1 Broché
ISBN 2 - 913329 - 25 - X Version électronique

Résumé

L'évolution des technologies, les exigences de productivité, l'accroissement de la complexité des circuits intégrés ont contribué à l'émergence des composants virtuels (IPs), ainsi qu'au développement de logiciels d'aide à la conception de circuits intégrés. L'utilisation de l'abstraction et des composants déjà conçus sont les clés de ces défis.

L'objet de cette thèse est le parcours des principaux niveaux d'abstraction de la synthèse matérielle, la synthèse logique, RTL et comportementale, en dégageant pour chacun d'entre eux les contraintes de conception qui vont devenir les critères de sélection d'un bloc réutilisable. Il ne reste plus qu'à concevoir un éventail de blocs dans une approche de réutilisation: les blocs doivent être facilement sélectionnables, puis paramétrables, et enfin intégrables dans un circuit plus important. La conception des blocs comportementaux, appliquée au codage correcteur d'erreur, nous amène à réfléchir sur les méthodologies de conception et de réutilisation des composants virtuels.

Abstract

Technological progress, designers' productivity expectations, exponential increase of design complexity, have lead to the emergence of virtual components (IPs), as well as the development of software for integrated circuits design automation. The use of abstraction and components already designed are the keys of these challenges.

The aim of this thesis is the study of the main abstraction levels of hardware synthesis, logic, RT and behavioral synthesis, in order to deduce, for each of them, the optimization constraints that will become the criteria of selection for a reuse block. Then, one has to design an array of blocks in a reuse approach: the blocks must be easily selected, parameterized, and then integrated in a complete circuit. The design of behavioral blocks, applied to the error correcting codes, gives us food of thought about the design and reuse methodologies of virtual components.

micro-électronique.

mots clés : conception, composants virtuels, IP, décomposition technologique, arithmétique, codage correcteur d'erreur.