Outil de CAO pour la génération d'opérateurs arithmétiques auto-contrôlables

I. Alzaher-Noufal

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

THESE

Pour obtenir le grade de

DOCTEUR DE L'INPG

Spécialité : Micro-électronique

préparée au laboratoire TIMA
dans le cadre de l'**Ecole Doctorale EEATS**

présentée et soutenue publiquement
par

Issam AlZAHER NOUFAL

Le 23 Mai 2001

Titre

Outils de CAO pour la Génération d'Opérateurs Arithmétiques Auto-Contrôlables

Directeur de thèse

Mihail NICOLAIDIS

JURY

M. René DAVID	, Président
M. Abbas DANDACHE	, Rapporteur
M. Christian DUFAZA	, Rapporteur
M. Mihail NICOLAIDIS	, Directeur de thèse

Abstract

With the increasing integration density, new generations of integrated circuits are becoming more and more sensitive to noise sources such like power line disturbances, electromagnetic influence, etc. Further, it is now proved that some particles coming from the space (alpha particles and atmospheric neutrons) can also interfere deep sub-micron circuits operation (soft errors).

Face to this challenge, self-checking techniques could provide good solutions. A good solution must of course result in reduced hardware overhead cost and little or no performance penalty. At that stage, two global difficulties appear. First, self-checking techniques are known by a restricted group of designers, and second, there are no professional CAD tools for automating self-checking circuits generation.

In this work, we show that self-checking multipliers based on residue codes can result in a very little hardware overhead, especially for large multipliers. In a second time, we generalize fault secure solutions for multipliers, adders and shift registers based on the parity code. The new versions have several parity bits in order to increase fault coverage.

We have implemented the presented solutions in a CAD tool developed by our research group. This tool offers many different self-checking arithmetic and logical operators to make flexible the construction of self-checking data paths.

Liste des Publications

1. "A CAD Framework for Efficient Self-Checking Data Path Design".
   R. O. Duarte, I. A. Noufal, M. Nicolaidis.
   3rd IEEE IOLTW, Crete, Greece, July 7-9 1997.

2. "A CAD Framework for Generating Self-Checking Multipliers Using residue Codes".
   I. A. Noufal, M. Nicolaidis.
   4th IEEE IOLTW, Capri, Italy, July 6-8 1998.

3. A CAD Framework for Generating Self-Checking Multipliers Based on residue Codes".
   I. A. Noufal, M. Nicolaidis.
   DATE 99, Munich, Germany, March 1999. **Best Paper Award**.

4. "Self-Checking Circuits versus Realistic Faults in Very Deep Submicron",
   L. Anghel, M. Nicolaidis, I. A. Noufal,
   Proceedings of VTS'00, pp.263-269, Montreal, May, 2000.

Remerciements

Je tiens à exprimer ma reconnaissance à

Monsieur René DAVID, du Laboratoire d'Automatique de Grenoble (LAG), pour l'honneur qu'il m'a fait en acceptant de présider le jury de cette thèse.

Messieurs Abbas DANDACHE de l'université de Metz et Christian DUFAZA de l'université de Montpellier II pour avoir accepté d'être rapporteurs de ce travail.

Messieurs Bernard COURTOIS, Directeur du laboratoire TIMA et Mihail NICOLAIDIS responsable du groupe Systèmes Intégrés Sûrs (RIS), pour m'avoir donné la possibilité de faire cette recherche.

Monsieur Stanisalw J. Piestrak de Politechniki Worclawskiej de Pologne pour sa précieuse collaboration.

Aux amis et collègues de TIMA pour les bons moments passés ensemble.

Table des Matières

Introduction

Avec la complexité de plus en plus croissante des circuits intégrés, le test au niveau fabrication s'avère une tache difficile et insuffisante. Plus de 90% des fautes occurrentes dans les systèmes électroniques sont temporaires [Sel68], et donc difficiles à détecter (voir indétectables) en effectuant des tests logiciels périodiques. La technique d'auto contrôle en ligne semble être une solution potentielle à ce problème. Cette technique permet de détecter des erreurs résultant de fautes permanentes ou transitoires.

Dans le passé, l'utilisation des circuits auto-contrôlables était restreinte aux applications exigeant un niveau élevé de sécurité (e.g. contrôle des chemins de fer [Duf96], contrôle des fonctions critiques dans l'automobile [Boe97], etc.), ou évoluant dans des environnements hostiles (e.g. satellites). Le rétrécissement des dimensions des composants, la réduction du niveau de l'alimentation et l'augmentation de la vitesse de fonctionnement liés à l'intégration à très grande échelle, affectent défavorablement les marges de bruit des circuits. Alors une protection en ligne contre les fautes devient inévitable même pour des applications où la fiabilité n'est pas cruciale [Nic98a], [Nic98b], [Nic98c]. Par exemple, pour les technologies CMOS de 0,1 μm ou moins, la fréquence des erreurs transitoires ou "single event upsets" dues aux neutrons "chauds" atmosphériques sera intolérable même à la surface de la terre. Avec les hautes fréquences de fonctionnement, ce phénomène affectera non seulement les mémoires mais aussi les parties logiques parce que la durée d'une faute transitoire devient comparable au cycle d'horloge. Le problème majeur va concerner les parties logiques puisque les mémoires peuvent être protégées efficacement en utilisant les codes de détection et de correction d'erreurs. Une solution potentielle à ce problème est l'auto-contrôlabilité qui permet de détecter immédiatement les erreurs si lieu est. Les erreurs pourront ensuite être corrigées en répétant la dernière opération.

Le peu d'applications utilisant la technique d'auto-contrôle en ligne a rendu quasi inexistant le développement d'outils de CAO permettant de générer des circuits auto-contrôlables. Le manque de ces outils augmente dramatiquement l'effort de conception de circuits auto-contrôlables. Pour cette raison, les industriels préfèrent recourir à des méthodes comme la duplication ou la triplication qui présentent un coût de développement moindre malgré leur coût de production relativement élevé. L'objectif de cette thèse est de penser à ce problème, et de développer de tels outils.

Le chapitre 1 de ce manuscrit est une présentation de la terminologie utilisée dans le domaine des circuits auto-contrôlables, ainsi qu'une exposition de l'état de l'art des chemins de données auto-contrôlables.
Ensuite, la première partie de notre travail est consacrée à l'étude et à l'implémentation d'additionneurs et de multiplieurs auto-contrôlables basés sur le code de résidu. Cette partie est réalisée dans les chapitres 2, 3 et 4. La seconde partie (chapitre 5) est une généralisation de schémas (additionneurs, multiplieurs et registres à décalage) sûrs en présence de fautes basés sur le code de parité. Les développements réalisés dans [Nic93] [Nic97a] [Nic97b] [Dua97a] concernent des additionneurs, des multiplieurs et des registres à décalage auto-contrôlables basés sur un seul groupe de parité. Notre travail consiste à généraliser et à implémenter les mêmes schémas mais avec plusieurs groupes de parité dans le but d'augmenter la couverture de fautes.

Nous avons vu que les outils de CAO dédiés aux circuits auto-contrôlables sont quasi inexistants. Une percée dans ce domaine est réalisée dans [Dua97c] [Ped95]. Plusieurs types de circuits auto-contrôlables ont été réalisés et intégrés dans la bibliothèque que possède l'outil développé. Par notre travail, cette bibliothèque s'est trouvée enrichie de tous les circuits auto-contrôlables étudiés dans cette thèse. L'outil de CAO sera présenté dans le chapitre 6.

Chapitre 1. Définitions et Etat de l'Art

1.1. Les Circuits Auto-Contrôlables

L'auto-contrôle en ligne vérifie le fonctionnement des circuits pendant leur mode normal d'opération. Les sorties d'un circuit dans un système sont imprévisibles, d'où le besoin d'une propriété invariante qui doit être vérifiée par tous les vecteurs de sortie. La technique d'auto-contrôle en ligne permet de détecter des erreurs en introduisant de la redondance. Un circuit complexe est divisé en ses blocs constituants et chacun de ces blocs est implémenté selon la structure de la figure 1.1. Les sorties d'un bloc appartiennent à un code de détection d'erreur qui impose ainsi une propriété invariante. Un contrôleur du code utilisé vérifie ensuite en ligne si l'invariance est respectée pour chaque vecteur de sortie.

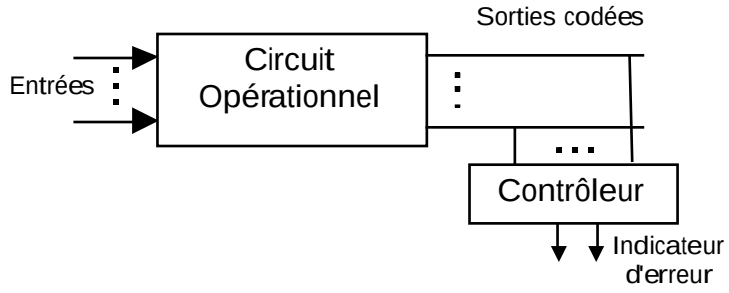


Figure 1.1. La structure générale d'un circuit auto-contrôlable

Par la suite nous allons présenter différentes classes de circuits auto-contrôlables définissant ainsi diverses propriétés de protection contre les fautes.

Définition 1 : Un circuit *G* est sûr en présence de fautes pour un ensemble de fautes *F*, si pour chaque faute $f \in F$, il ne produit jamais un mot incorrect appartenant au code sortie.

La propriété "sûr en présence de fautes" garantie que le circuit *G* ne génère pas des sorties incorrectes pour des fautes dans *F*. Imaginons maintenant la situation où une faute demeure indétectable dans *G* et supposons que survienne une nouvelle faute. L'ensemble des deux fautes n'est plus dans *F* et le circuit peut perdre sa propriété "sûr en présence de fautes". D'autres conditions sont donc nécessaires pour éviter cette situation.

Définition 2 : Un circuit *G* est auto-testable pour un ensemble de fautes *F*, si pour toute faute dans *F*, il existe au moins un vecteur d'entrée se produisant pendant le mode normal de fonctionnement qui permet de la détecter [Car68].

Définition 3 : Un circuit *G* est totalement auto-contrôlable pour un ensemble de fautes *F*, s'il est sûr en présence de fautes **et** auto-testable pour *F*.

Avec la propriété sûr en présence de fautes, il est garanti que la première faute survenante génère toujours des erreurs détectables. Ensuite, supposant que entre l'occurrence de deux fautes s'écoule suffisamment de temps pour que le circuit reçoive toutes les entrées permettant de tester ses fautes internes, la propriété auto-testable garantie que la première faute est détectée avant qu'une autre faute survienne dans le circuit auto-contrôlable.

La propriété la plus difficile et essentielle à réaliser est la sûreté en présence de fautes car elle garantie la détection de toutes les erreurs produites en cas d'occurrence d'une première faute. L'occurrence d'une première faute indétectable suivie d'une deuxième faute dont la combinaison avec la première produit des erreurs indétectables est une situation bien moins probable, et beaucoup d'applications ne nécessitent une protection aussi renforcée. Pour ces raisons, la première étude se concentre sur la conception de circuits sûrs en présence de fautes. De plus, pour les cas de collages logiques, une faute indétectable correspond à une redondance logique. Ainsi, tout collage logique redondant est évité en générant des circuits sans redondance. Les circuits arithmétiques étudiés dans notre travail sont des circuits minimisés et ne contiennent pas de redondance. Ils ne peut donc pas y avoir des fautes de collage logique indétectables.

Notons aussi que seule la propriété "sûr en présence de fautes" s'applique au cas des fautes transitoires. Ces fautes n'étant pas permanentes ; la propriété "auto-testable" ne s'applique pas dans leur cas. Des techniques de scroutage périodiques sont souvent employées pour réduire la latence d'éventuels états erronés stockés dans les parties de mémorisation des circuits.

Finalement, pour certains autres types de fautes, tels que les courts-circuits par exemple, il n'est pas toujours aisé d'éliminer toute faute indétectable. Dans ce cas et pour des applications nécessitant de très hauts niveaux de sécurité on pourrait imposer certaines contraintes sur la structure du circuit de façon à garantir qu'une faute indétectable ne brise pas la propriété "sûr en présence de fautes" [Nic87]. Ces techniques implémentent la propriété suivante :

Définition 4 : Un circuit *G* est fortement sûr en présence de fautes pour un ensemble de fautes *F*, si pour chaque faute *f* dans *F*, ou bien
a) il est totalement auto-contrôlable ou bien
b) en présence de *f* il préserve la sûreté en présence de fautes, et si une nouvelle faute appartenant à F survient, alors pour la faute combinée soit (a) ou soit (b) [Dav78] [Smi78].
Les circuits fortement sûrs en présence de fautes constituent la classe la plus large de circuits satisfaisant la propriété totalement auto-contrôlable.

Les contraintes additionnelles nécessaires pour garantir la propriété d'auto-testabilité ou de circuit fortement sûr en présence de fautes, dans le cas de défauts spécifiques tels que les courts-circuits, impliquent un coût matériel supplémentaire qui n'est pas justifié pour une majorité d'applications, étant donné la faible probabilité d'occurrence des situations contre lesquelles ces contraintes protègent le circuit. **Par conséquent, dans le reste de ce rapport on considère uniquement la propriété "sûr en présence de fautes"**.

1.2. Les Codes de Détection d'Erreurs

La manière la plus évidente pour parvenir à la sûreté en présence de fautes est de dupliquer le circuit en question et d'utiliser ensuite un comparateur pour tester l'égalité des sorties.
Cette technique introduit un surcoût supérieur à 100%. Pour cette raison, d'autres techniques ont été élaborées pour réduire ce surcoût. Ces techniques emploient des codes de détection d'erreurs ayant un coût inférieur à celui du code de duplication. Quelques codes parmi les plus utilisés dans l'autocontrôle de circuits intégrés logiques sont présentés par la suite.

1.2.1. Le Code de Parité

Le code de parité est le code de détection d'erreur le plus simple et le moins coûteux. Il consiste à ajouter un bit supplémentaire aux bits d'information à vérifier. Ce bit représente la parité du nombre de "uns" dans le mot d'information considéré. Le code de parité permet de détecter des erreurs sur un seul bit ou plus généralement sur un nombre impair de bits.

1.2.2. Le code "double-rail"

Ce code consiste à dupliquer et à complémenter la partie information. Il permet de détecter des erreurs multiples mais introduit une redondance maximale similaire à la duplication (le nombre des bits de contrôle est égal au nombre des bits d'information).

1.2.3. Les Codes Non Ordonnés

Dans ce type de codes il n'y a pas deux mots de code différents *x* et *y* tels que *x* "couvre" *y* (ou *x* > *y*), où *x* couvre *y* veut dire que *x* a 1 à chaque position de bit où *y* a 1. Par exemple si *x* = 10010 et *z* = 10001, alors ni *x* couvre *y* ni *y* couvre *x*, et dans ce cas *x* et *y* peuvent appartenir un code non ordonné. A partir de cette propriété, si une erreur multiple contenant seulement des erreurs du type $1 \rightarrow 0$ affecte un mot de code *x*, alors le résultat *z* sera couvert par *x* et il ne peut pas appartenir au code non ordonné. Ainsi cette erreur sera détectée. De manière similaire, si une erreur multiple contenant seulement des erreurs du type $0 \rightarrow 1$ affecte un mot de code *x*, alors le résultat *z* va couvrir *x* et l'erreur sera encore détectée. Un code non ordonné détecte donc toutes les erreurs unidirectionnelles. Les codes non ordonnés les plus intéressants sont le code m-out-of-n et le code de Berger.

1.2.3.1. Le Code *m*-parmi-*n* (*m*-out-of-*n*)

Le code *m*-out-of-*n* est un code non séparable (les bits d'information et les bits de contrôle sont mélangés). Ce code est composé de mots de code qui ont m 1's. (e.g. le code 2-parmi-4 : 1100, 0101, 1001 etc.). Ce code est un code non ordonné non séparable optimal (redondance minimale pour le codage non ordonné). Pour un *n* donné, le nombre de mots de code de poids *m* est donné par *n*!/(*n*-*m*)! *m*!. Ce nombre est maximal pour le code *n*/2-parmi-*n*. Ainsi, les codes *n*/2-parmi-*n* supportent le taux d'information le plus élevé par rapport à tous les codes de détection d'erreurs unidirectionnelles.

1.2.3.2. Le Code de Berger

Ce code est un code non ordonné séparable. La partie de contrôle de ce code représente le nombre de 0's dans la partie d'information. Par exemple, pour la partie information *I* = 100101 la partie contrôle est *C* = 011. Dans une seconde variance de ce code, la partie de contrôle sera égale au complément du nombre de 1's dans la partie d'information. Par exemple, pour la partie information *I* = 100101 la partie de contrôle est *C* = 100. Pour n bits d'information le nombre de bits contrôle est égal à $\lceil \log_2 (n+1) \rceil$.

1.2.4. Les Codes Résidus

Ces codes sont divisés en deux catégories : séparables et non séparables. Dans les codes arithmétiques séparables avec une base A, les mots de code sont obtenus en associant à la partie information X une partie test $X' = |X|_A$ (le code résidu), ou $X' = A\text{-}|X|_A$ (le code résidu inverse).

Dans les codes arithmétiques non séparables avec une base *A*, les mots de code sont égaux au produit des mots d'origine (non codés) par la base *A*.

Les codes arithmétiques sont intéressant dans le cas de test des opérations arithmétiques, parce qu'ils sont préservés pour de telles opérations. Les codes arithmétiques les plus utilisés sont les codes séparables qui permettent une implémentation à faible coût, où la base de détection d'erreur est de forme $A = 2^m-1$. Les codes arithmétiques de coût minimum permettent de détecter des erreurs arithmétiques dont la valeur dépend de la base *A*. Pour $A = 3$, qui correspond à deux bits de contrôle, le code détecte toutes les erreurs arithmétiques simples (i.e., la valeur arithmétique de l'erreur est une puissance de 2).

La conception de circuits sûrs en présence de fautes est une tâche difficile. Pour un code de détection d'erreurs donné nous devons garantir la propriété suivante : chaque faute modelée affectant le circuit crée des erreurs locales qui doivent se propager vers les sorties primaires comme erreurs détectables par le code.

Le choix du code est une étape critique. Un code possédant une grande capacité de détection d'erreurs rend plus facile l'accès à la sûreté en présence de fautes, mais augmente le nombre de sorties introduisant ainsi un surcoût élevé. D'autre part, un code avec une faible capacité de détection d'erreurs introduit moins de sorties mais exige des modifications significatives de la structure du circuit (qui sont synonymes de surcoût).

L'idéal pour un système logique donné sera d'appliquer un seul code de détection d'erreur. L'utilisation d'un seul code permet de diminuer le nombre de codeurs et de contrôleurs dans le système. Malheureusement, ceci n'est pas facile à réaliser car chaque partie du système a un code qui lui est le mieux adapté et l'utilisation d'un autre code peut être inefficace et introduire un surcoût très élevé.

Prenons par exemple les mémoires. Vu la structure en tranches de la mémoire, une faute dans une cellule génère une seule erreur sur les bits d'information. Par conséquent, les mémoires sont contrôlées en utilisant le code de parité qui introduit dans ce cas un surcoût minimum par rapport aux autres codes de détection d'erreur. De même, le code de parité est utilisé pour tester les bus et les registres. D'un autre côté il peut être plus intéressant d'implémenter un multiplieur en utilisant un code arithmétique au lieu d'un code de parité. Les outils que nous devons développer permettent de générer automatiquement les différents blocs et d’estimer ainsi le coût des différentes solutions afin de choisir la plus intéressante.

1.3. Le Modèle de Fautes

La conception de circuits auto-contrôlables passe tout d'abord par la détermination d'un modèle de fautes pour lequel on développe des techniques permettant d'assurer les différentes propriétés des circuits auto-contrôlables.

Le modèle de fautes le plus souvent utilisé est le modèle de collage-à-z , $z \in \{0, 1\}$. Le modèle en question suppose qu'une faute est une ligne à l'état logique z indépendamment des valeurs appliquées aux entrées du circuit. Ce modèle est le plus simple et permet de donner une très bonne couverture de fautes dans la plupart des circuits numériques. Toutes les analyses faites dans cette thèse couvrent ce modèle.

1.4. Conception de Contrôleurs

La fonction d'un circuit contrôleur est de signaler l'occurrence d'erreurs à son entrée par un signal d'erreur et l'occurrence d'entrées correctes par un signal de bon fonctionnement. L'ensemble des mots de code de sortie indiquant un bon fonctionnement constitue l'espace de code de sortie du contrôleur et l'ensemble des mots indiquant une erreur constitue l'espace de non-code de sortie. D'après sa fonction le contrôleur doit vérifier la propriété suivante :

Définition 5 : Un contrôleur est à codes disjoints (*CD*) s'il associe aux mots de code d'entrée des mots de code de sortie et aux mots non-code d'entrée des mots non-code de sortie.

Si une faute *f* demeure indétectable dans un contrôleur *G*, elle peut masquer une erreur produite dans le circuit en amont. Pour éviter cette situation, *G* doit être auto-testable (définition 2).

Définition 6 : Un circuit *G* est un contrôleur auto-testable pour un code *C* et pour l'ensemble de fautes *F* s'il est *CD* pour *C* et auto-testable pour *F* [And73].

En supposant le temps entre l'occurrence de deux fautes suffisamment large, une faute dans un contrôleur *CD* et auto-testable est toujours détectée avant l'apparition d'une autre faute.

Un contrôleur doit avoir au moins deux sorties pour signaler l'erreur. Si le contrôleur a une seule sortie qui a la valeur *z* s'il n'y a pas d'erreur et $\bar{z}$ en cas d'erreur, $z \in \{0,1\}$, une erreur forçant sa sortie à la valeur *z* ne sera jamais détectée et le contrôleur perd sa fonction. Par conséquent, l'augmentation du nombre des sorties du contrôleur augmente sa fiabilité. On utilise en fait des contrôleurs à deux sorties dont les valeurs 01 et 10 indiquent un fonctionnement correct, tandis que les valeurs 00 et 11 sont synonymes d'erreur.

Une catégorie de circuits appelés les circuits fortement *CD* a été introduite dans [Nic88]. Ces circuits constituent la classe la plus large des circuits *CD* vérifiant la propriété "totalement auto-contrôlable", puisqu'ils préservent la propriété *CD* même en présence de fautes internes indétectables de *F*.

Définition 7 : [Nic88] Un circuit est fortement *CD* pour un ensemble de fautes *F* si avant l'occurrence de fautes il est *CD*, et pour chaque faute dans *F* on a :
a) le circuit est auto-testable ou
b) le circuit associe les mots d'entrée qui ne sont pas des mots de code aux mots de sortie qui ne sont pas des mots de code, et si une autre faute de *F* survient, alors soit (a) soit (b) est vraie pour la combinaison de fautes.

La conception d'un contrôleur auto-testable est une tâche difficile parce que toutes les fautes internes doivent être détectées en n'utilisant que les mots de code à l'entrée. Heureusement, pour les codes qui nous intéressent et qui sont les plus utilisés, il existe déjà des contrôleurs auto-testables qui ont été un sujet d'étude pour de nombreux chercheurs dans le passé.

1.4.1. Les Contrôleurs de Parité

Un arbre de parité calcule la parité de ses entrées et peut être utilisé comme un contrôleur de parité. Ce circuit est testable facilement puisque, 4 vecteurs d'entrée bien sélectionnés sont suffisant pour tester un arbre composé de portes EXOR à deux entrées. Toutefois, ce circuit a une sortie unique et l'application de mots de code à l'entrée (i.e., seulement des mots dont la parité est paire, ou seulement des mots dont la parité est impaire) ne détecte qu'une seule des deux fautes de collage de la sortie (voir figure 1.2.a). Un contrôleur de parité à deux sorties peut être réalisé en divisant les entrées en deux groupes et en utilisant un arbre de parité pour chaque groupe (voir figure 1.2.b). Comme les entrées sont codées en parité paire et la sortie d'un arbre inversée, les valeurs de sortie 10 et 01 indiquent un fonctionnent correct, tandis que les valeurs de sortie 00 et 11 indiquent la présence d'erreur.

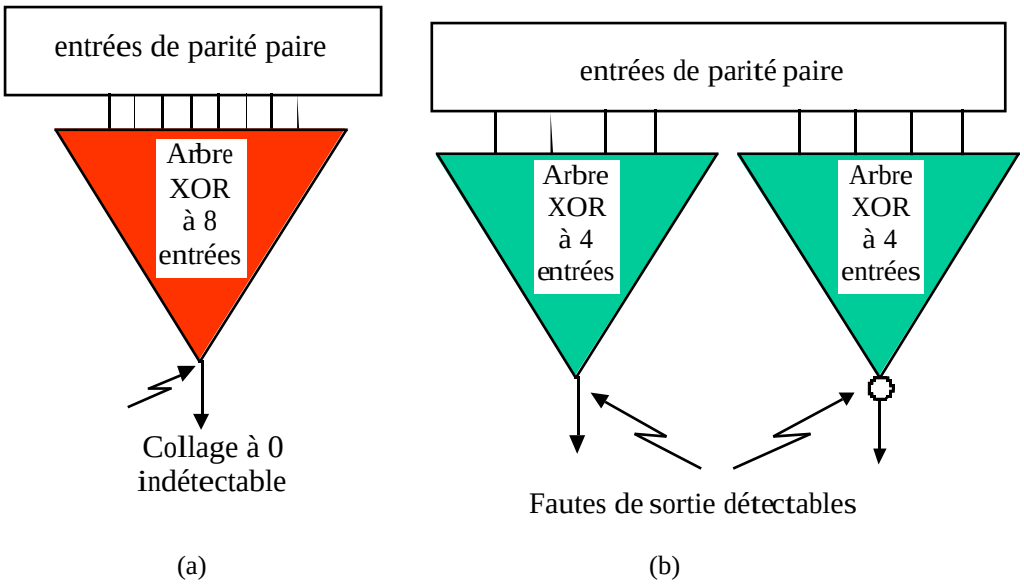
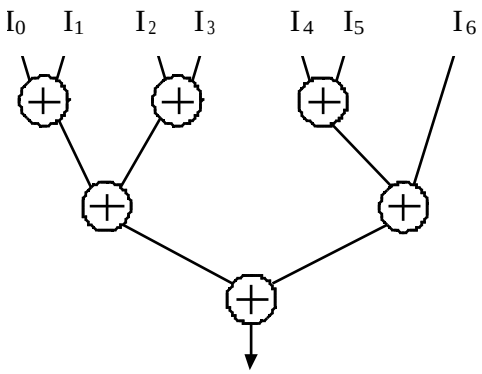


Figure 1.2. a) contrôleur de parité non auto-contrôlable
b) contrôleur de parité auto-contrôlable

Quatre mots de code bien choisis sont aussi suffisants pour tester ce contrôleur [And71]. La figure 1.3 montre un tel exemple. Un contrôleur de parité à 14 entrées est implémenté en utilisant deux arbres XOR à 7 entrées chacun. L'un de ces arbres est montré dans la figure 1.3. Les 14 entrées du contrôleur reçoivent uniquement des valeurs codées en parité paire (2^{13} valeurs possibles) ou uniquement des valeurs codées en parité impaire (2^{13} valeurs possibles). Néanmoins, chacun des arbres XOR à sept entrées composant le contrôleur reçoit toutes les combinaisons d'entrée possibles (2^7 valeurs possibles). Il est donc exhaustivement testable. De plus, quatre seulement des 2^{13} valeurs d'entrée sont suffisantes pour tester exhaustivement toutes les portes XOR de l'arbre. Un tel exemple des valeurs d'entrée est donné en figure 1.3.



I_0 I_1	I_2 I_3	I_4 I_5	I_6
00	00	00	0
01	01	01	0
10	11	10	1
11	10	11	1

Figure 1.3. Vecteurs de test pour un arbre XOR à sept entrées

1.4.2. Les Contrôleurs Double-Rail

Un contrôleur double-rail auto-testable peut être réalisé comme un arbre de parité en remplaçant les portes logiques XOR par la cellule de base du contrôleur double-rail. La cellule double-rail est présentée dans la figure 1.4. On peut remarquer que $F_0 = A \quad B$ et $F_1 = \overline{A \quad B}$. Cette propriété a été exploitée dans [Nic93] pour effectuer en même temps la génération de parité et le contrôle double-rail.

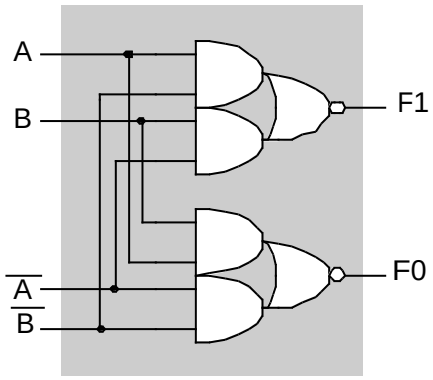


Figure 1.4. Cellule de base du contrôleur double-rail

La figure 1.5 représente le schéma d'un contrôleur double-rail à 8 bits.

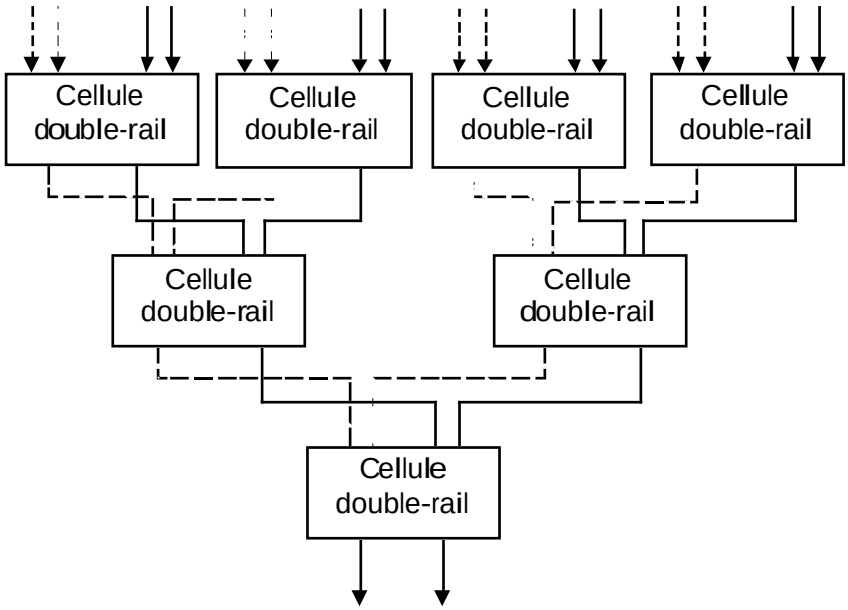


Figure 1.5. Contrôleur double-rail à 8 bits

Le contrôleur double-rail est testable facilement, puisque quatre mots de code sont suffisants pour tester un contrôleur double-rail de n'importe quelle taille [And71]. Ce contrôleur double-rail est très important dans la conception des circuits auto-contrôlables. Il peut être utilisé pour le contrôle des blocs duaux et aussi des blocs dupliqués en complémentant les sorties d'un des deux. Une autre utilisation importante de ce contrôleur consiste en la compression des différents signaux d'indication d'erreur fournis par les différents contrôleurs dans un circuit auto-contrôlable complexe. Comme nous l'avons vu, chaque contrôleur fournit une paire de signaux codés en double-rail. Ensuite, un contrôleur double-rail peut compresser les différentes paires fournies par les sorties des contrôleurs du système en une seule paire codée en double-rail. Cette paire constitue le signal d'indication d'erreur du système tout entier.

1.4.3. Les Contrôleurs du Code *m*-parmi-*n*

Plusieurs méthodes de réalisation de contrôleurs *m*-parmi-*n* ont été proposées dans la littérature [And73][Pas88]. Les sorties d'un contrôleur *m*-parmi-*n* sont au nombre de deux. Si le nombre de 1's à l'entrée du contrôleur est supérieur ou inférieur à m, alors la sortie est (0,0) ou (1,1) sinon elle est (1,0) ou (0,1). La figure 1.6 montre un exemple de contrôleur *1*-parmi-*4*.

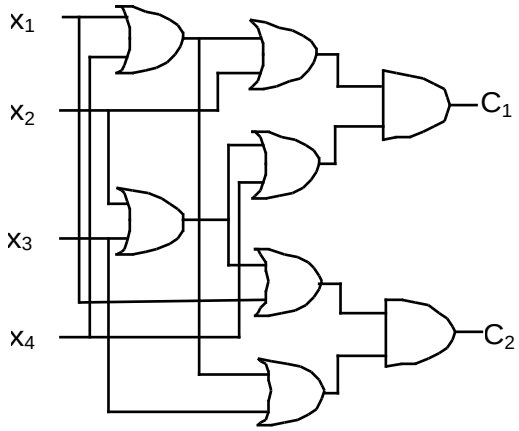


Figure 1.6. Un Contrôleur *1*-parmi-*4*

1.4.4. Le Contrôleur du Code de Berger

Nous avons vu que le code de Berger est un code séparable. L'implémentation la plus fréquente des contrôleurs des codes séparables est d'utiliser un bloc générateur qui reçoit comme entrée la partie information des mots de code et génère la partie contrôle. Ensuite, un contrôleur double-rail est utilisé pour comparer cette partie contrôle à la partie contrôle attachée au mot de code. Lorsque toutes les combinaisons binaires possibles apparaissent sur la partie information, cette structure de contrôleur est auto-testable pour des fautes affectant le générateur sachant que ce contrôleur n'est pas redondant. Le bloc générateur de la partie contrôleur est réalisé en utilisant un compteur de 1's. Ce compteur est composé de cellules de "full adders" et "half adders" disposées dans un arbre de Wallace comme ceux utilisés dans différentes structures de multiplieurs.

1.4.5. Les Contrôleurs des Codes Résidus

Les codes arithmétiques souvent utilisés dans les circuits auto-contrôlables sont les codes séparables et de coût minimum. Les circuits de contrôle et de génération de ces codes seront étudiés en détail dans le chapitre 2.

1.5. Chemins de Données Auto-Contrôlables

Les chemins de données sont des parties logiques essentielles (et généralement les plus grandes) dans les microprocesseurs et les microcontrôleurs. L'utilisation de chemins de données auto-contrôlables efficaces aura donc un impact important sur la conception des circuits intégrés futurs. Les circuits auto-contrôlables sûrs en présence de fautes garantissent la détection des erreurs générées à la sortie par les fautes permanentes ou temporaires. Les

conceptions vérifiant la sûreté en présence de fautes permettront donc d'obtenir des niveaux élevés de fiabilité.

Les inconvénients majeurs liés à la conception des circuits auto-contrôlables sont le surcoût et le grand effort de conception. L'utilisation des codes de parité pour tester les mémoires et les registres garantit la sûreté en présence de fautes tout en introduisant un faible surcoût. Par contre, les opérateurs arithmétiques peuvent produire des erreurs de sortie dont la multiplicité est aléatoire et donc généralement indétectables par les codes de parité. Les schémas de prédiction de parité [Sel68], [Gar68] ne permettent donc pas d'obtenir la sûreté en présence de fautes pour ces circuits. De récents développements [Nic93] [Nic97a] [Nic97b] [Dua97a] sur la conception d'opérateurs arithmétiques basés sur la prédiction de parité, ont présenté des solutions sûres en présence de fautes pour plusieurs blocs de base utilisés dans les chemins de données et parmi lesquels : additionneurs, ALUs, multiplieurs, diviseurs, registres à décalage et bandes de registres. Pour certains de ces blocs (additionneurs, ALUs, registres à décalage, bandes de registres) le surcoût matériel reste relativement faible, mais pour d'autres (multiplieurs, diviseurs) il devient plus élevé (de l'ordre de 45%). La figure 1.6 représente un exemple de chemin de données auto-contrôlable basé sur la prédiction de parité.

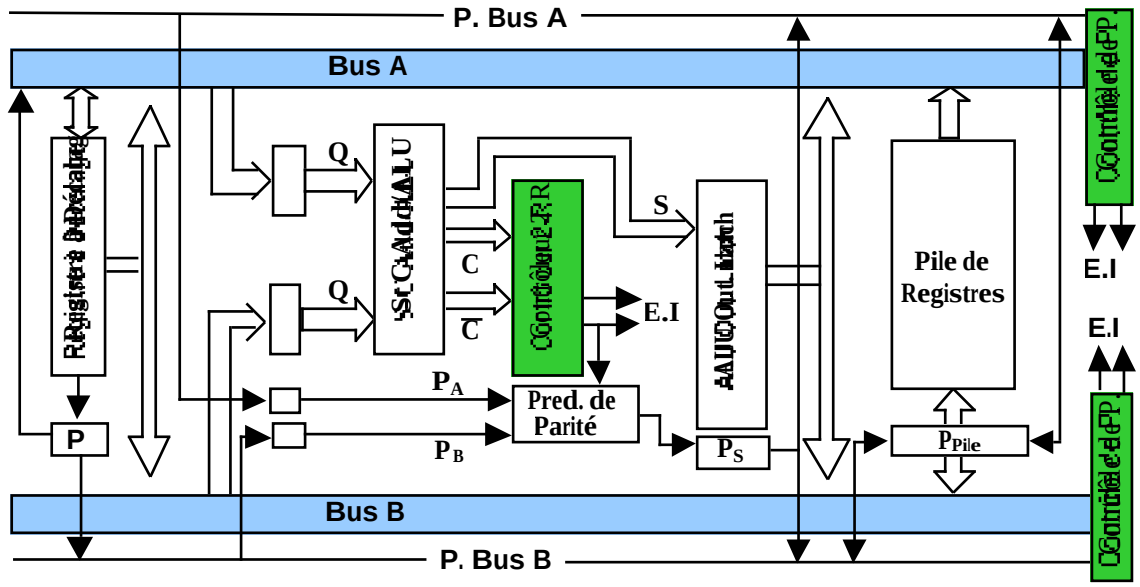


Figure 1.6. Chemin de données auto-contrôlable basé sur la prédiction de parité

Ce chemin de données comprend :

- Un bloc additionneur/ALU auto-contrôlable basé sur la technique présentée dans [Nic93] [Nic97b] et qui comprend le contrôle des retenues et la prédiction de parité.
- Un registre à décalage auto-contrôlable basé sur la prédiction de parité et implémenté selon [Dua97a].

- Une pile de registres (register file) incluant un bit de parité pour chaque registre.

Les différents blocs communiquent via les bus de données A et B. Les données sont contrôlées pendant leur transfert d'un bloc à un autre par les deux contrôleurs de parité connectés aux bus A et B.

Toutefois, les schémas de prédiction de parité proposés pour les multiplieurs ont un surcoût élevé (entre 40% et 50%) par rapport aux autres types de blocs. Dans le cas des multiplieurs de grandes tailles, ceci augmentera considérablement le coût du chemin de données tout entier. D'autres techniques moins coûteuses seront donc nécessaires pour obtenir des multiplieurs auto-contrôlables. Néanmoins, pour intégrer dans un chemin de données contrôlé par la parité, un multiplieur contrôlé par un autre code, on doit prévoir le contrôleur du multiplieur (qui ne peut plus être contrôlé par les contrôleurs de parité placés sur les Bus), ainsi qu'un générateur de parité pour fournir au reste du chemin de données la parité des résultats produits par le multiplieur.

Chapitre 2. Les Générateurs de Résidus

Dans ce chapitre nous présentons la structure générale des circuits arithmétiques auto-contrôlables basés sur le code résidu et nous détaillerons ensuite les générateurs de résidus qui constituent une composante de base de ces circuits.

2.1. Structure Générale

Un circuit arithmétique auto-contrôlable basé sur le code résidu a la structure générale de la figure 2.1.

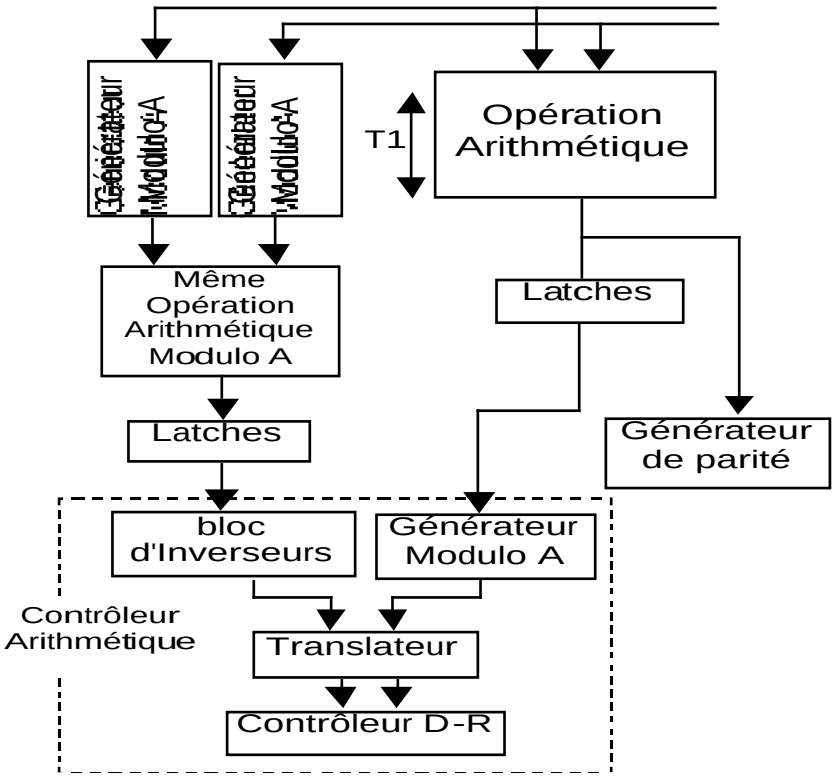


Figure 2.1. Structure générale d'un circuit auto-contrôlable basé sur le code résidu

Cette structure est composée des éléments suivants :

- Le bloc effectuant l'opération arithmétique
- Deux générateurs modulo *A* pour générer les bits de contrôle des opérandes d'entrée
- Un bloc qui effectue la même opération arithmétique modulo *A* pour générer les bits de contrôle du résultat de l'opération
- Un contrôleur arithmétique pour vérifier le résultat de l'opération et
- Un générateur de parité pour générer la parité du résultat de l'opération et qui sera utilisée par les autres blocs du chemin de données.

Le contrôleur arithmétique est constitué d'un générateur modulo *A* pour générer le résidu du résultat de l'opération, d'un bloc d'inverseurs, d'un translateur pour résoudre le problème de la double représentation du zéro (qui sera développé plus loin) et d'un contrôleur double-rail fournissant le signal d'erreur.
Pour un modèle de fautes donné, ce schéma est sûr en présence de fautes si la condition suivante est vérifiée : le résidu modulo *A* de la valeur arithmétique d'une erreur générée par une faute n'est pas nul, quelque soit la faute appartenant à l'ensemble de fautes modelées.
Le choix de la base A passe donc tout d'abord par une analyse des conséquences des fautes considérées dans l'opérateur arithmétique en question. Cette analyse sera faite pour chacun des opérateurs arithmétiques étudiés dans les chapitres suivants.
Par la suite nous allons d'abord présenter les générateurs de résidu qui constituent un élément important dans le schéma présenté ci-dessus, et ensuite nous parlerons du translateur.

2.2. Les Générateurs de Résidus

Le générateur de résidus constitue un bloc de base dans les circuits auto-contrôlables basés sur le code résidu. Plus les dimensions de ce générateur sont grandes, plus le coût de l'auto-contrôlabilité est élevé. Il est donc naturel de chercher à utiliser les générateurs de résidus de coûts faibles. Par la suite, une étude détaillée sur les générateurs de résidus existants est réalisée.
On considère un générateur modulo *A* à *n* entrées $\{x_{n-1}, \cdots, x_1, x_0\}$ et *a* sorties $\{s_{a-1}, \cdots, s_1, s_0\}$, où $a = \lceil log_2 A \rceil$. Ce générateur produit pour chaque entier $X = \sum_{j=0}^{n-1} 2^j x_j$ la valeur $|X|_A = \sum_{j=0}^{a-1} 2^j s_j$, i.e., il calcule :

$$|X|_A = \left| \sum_{j=0}^{n-1} 2^j x_j \right|_A \qquad (1)$$

Le schéma le plus évident pour calculer $|X|_A$ est la division de *X* par *A*, mais cette technique est très lente et coûteuse. Une meilleure méthode est basée sur l'algorithme de [Sza69] qui utilise l'expression suivante :

$$|X|_A = \left| \sum_{j=0}^{n-1} \left| 2^j \right|_A x_j \right|_A \qquad (2)$$

Si on dispose des valeurs $|2^j|_A$, $|X|_A$ peut être calculé en additionnant modulo *A* les termes $|2^j|_A$ pour lesquels $x_j = 1$. La somme est obtenue en utilisant un arbre de *n-1* additionneurs modulo *A*. Le schéma résultant est régulier mais coûteux et lent.
Prenons maintenant le cas spécial où $A = 2^a - 1$. Soit $b = \lceil n/a \rceil$ le nombre d'octets B_i, de *a* bits chacun, existant dans *X*. En remarquant que $|2^{t-a}|_{2^a-1} = 1$ (*t* entier positif), on peut démontrer l'égalité suivante :

$$\left|X\right|_{2^a-1} = \left| \sum_{i=0}^{b-1} b_i 2^{ai} \right|_{2^a-1} = \left| \sum_{i=0}^{b-1} b_i \right|_{2^a-1} \qquad (3)$$

où b_i est la valeur décimale de l'octet B_i avec $0 \le b_i \le 2^a - 1$. Le calcul de $\left|X\right|_{2^a-1}$ peut alors se faire avec un arbre de *b-1* additionneurs modulo $2^a - 1$. Un additionneur modulo $7 = 2^3 - 1$ de deux résidus modulo *7* (R_1 et R_2) est présenté par la figure 2.2.

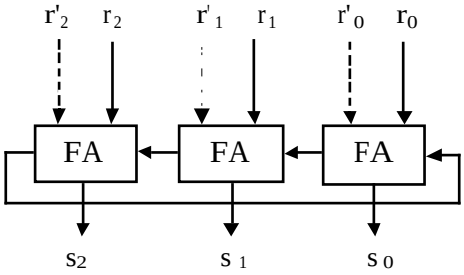


Figure 2.2. Un additionneur modulo 7 de deux résidus modulo 7

Ce schéma n'est autre que celui d'un additionneur séquentiel et la cellule FA est la cellule d'addition de base.
Puisque $|2^3|_7 = 2^0$, la retenue sortante est rebouclée pour être additionnée avec les bits de poids zéro. A partir du bloc de base présenté dans figure 2.2, on peut construire un générateur de résidus modulo 7 de n'importe quelle taille. La figure 2.3 représente le schéma d'un tel générateur pour 16 bits.
Les générateurs de résidus pour les valeurs du type $A=2^a-1$ sont moins complexes et moins coûteux que pour des valeurs de A différentes. Les codes arithmétiques avec des bases de détection d'erreurs de la forme 2^a-1 sont alors appelés les codes arithmétiques de coût minimum (*low cost arithmetic codes*).

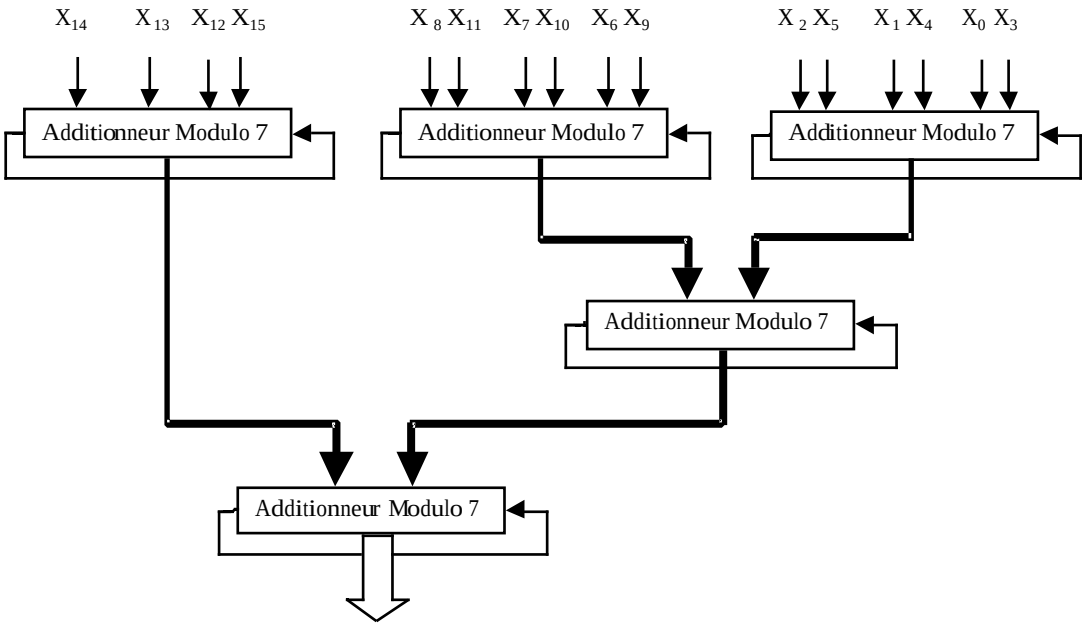


Figure 2.3. Générateur Modulo 7 à 16 bits

Dans [Pie94], de nouvelles méthodes pour la conception de générateurs de résidus sont exposées. Les techniques présentées sont valables pour tout A impair, mais les valeurs du type $A = 2^a-1$ donnent toujours les schémas les plus simples. Les générateurs de [Pie94] sont ceux que nous avons implémenté puisqu'ils s'avèrent les plus rapides. Par la suite, nous allons exposer ces différents générateurs selon les valeurs de A et de n.
Les définitions suivantes sont de base :

Définition 1 : la période *P(A)* d'un nombre impair *A* est la distance minimale entre deux différents 1's dans la séquence des résidus des puissances de *2* calculés modulo *A*, i.e. : $P(A) = min \{j \mid j>0 \text{ et } |2^j|_A = 1\}$.

Exemple : $|2^0|_9=1$, $|2^1|_9=2$, $|2^2|_9=4$, $|2^3|_9=8$, $|2^4|_9=7$, $|2^5|_9=5$, $|2^6|_9=1$. Alors $P(9)=6$.

Définition 2 : la demi-période *HP(A)* d'un nombre impair *A* est la distance minimale entre les valeurs *1* et *A-1* dans la séquence des résidus des puissances de *2* calculés modulo *A*. (Rq : $(A-1) \bmod A \equiv -1$). Exemple $HP(9)=3$.

La période *P(A)* existe pour tout nombre impair *A*. Si la demi-période existe pour un nombre impair *A*, on a : $P(A) = 2HP(A)$. *P(A)* et *HP(A)* peuvent être calculées par la formule récursive suivante :

$|2^i|_A = |2 \cdot |2^{i-1}|_A|_A$.

Pour certaines valeurs de A, on a les formules suivantes :

- $HP(2^a-1)$ n'existe pas alors que $P(2^a-1) = a$.
- $HP(2^{a-1}+1) = (a-1)$ et $P(2^{a-1}+1) = 2(a-1)$. Pour les bases de la forme $2^{a-1}+1$, la demi-période sera donc utilisée pour réaliser le générateur de résidu.

Per(A) est définie par $Per(A) = min \{P(A), HP(A)\}$. $Per(A)= HP(A)= P(A)/2$ pour tout nombre impair pour lequel *HP(A)* existe.

2.2.1. Générateur Modulo A Utilisant P(A)

Dans ce cas on suppose $per(A) = P(A)$ est assez petit et que le rapport $n/P(A)$ est assez grand. Les bits $x_j, x_{P(A)+j}, x_{2P(A)+j}, \ldots$ $(0 \le i \cdot P(A) + j < n)$ donnent la même valeur de résidu $|2^j|_A$ $(0 \le j \le P(A)-1)$. Le générateur modulo A est réalisé selon la procédure suivante :

Procédure 1 :

1. Partition de l'ensemble des bits d'entrée en sous-ensembles $G_j = \{x_q \mid q = i.P(A) + j\}, \ 0 \le j \le P(A)-1$.
2. Si $n \le 2P(A) + 1$ on passe à l'étape 3 et sinon on réduit les *n* bits en *n'* bits utilisant des additionneurs sans propagation de retenue ou "Carry Save Adders" (*CSAs*). Pour $n' = 2P(A)$ les sous-ensembles G_j (résultant de la dernière partition) ont chacun 2 bits et pour $n' = (2P(A) + 1)$, $G_{P(A)}$ a 3 bits et les autres sous-ensembles ont chacun 2 bits.
3. On réduit les *n'* bits en $n'' = P(A) + 1$ bits en utilisant un additionneur séquentiel ou "Carry Propagate Adder" (*CPA*). Si $A = 2^a-1$, $n'' = P(A)$ et dans ce cas la retenue sortante est rebouclée.
4. Calcul du résultat final (le résidu) en utilisant une *ROM* ou un bloc logique.

Exemple 1 : Conception d'un générateur modulo 21 et d'un générateur modulo 7

La figure 2.4 représente le schéma du générateur modulo *21* pour un vecteur d'entrée de 16 bits conçu selon la procédure 1. Les *16* bits d'entrée sont divisés en $P(21) = 6$ sous-ensembles : $G_0 = \{x_0, x_6, x_{12}\}$, $G_1 = \{x_1, x_7, x_{13}\} \cdots$, $G_5 = \{x_5, x_{11}\}$. Les cellules *FA* et *HA* sont les cellules d'addition de base. Chaque *FA* et *HA* dans le schéma additionne des bits dont le résidu de leur valeur décimale est le même (par exemple *1*, *64*, 64^2 ont le même résidu modulo *21*, et donc x_0, x_6 et x_{12} sont additionnés ensemble). Une étape de *CSA* permet de réduire les $n = 16$ bits en $n' = 12$ bits. Ces 12 bits sont ensuite réduits en $n'' = 7$ bits en utilisant un *CPA*. Le résultat final est calculé par un bloc logique implémentant l'équation : $(y_0 + y'_0 + y_1 + y_2 + y_3 + y_4 + y_5) \bmod 21$.

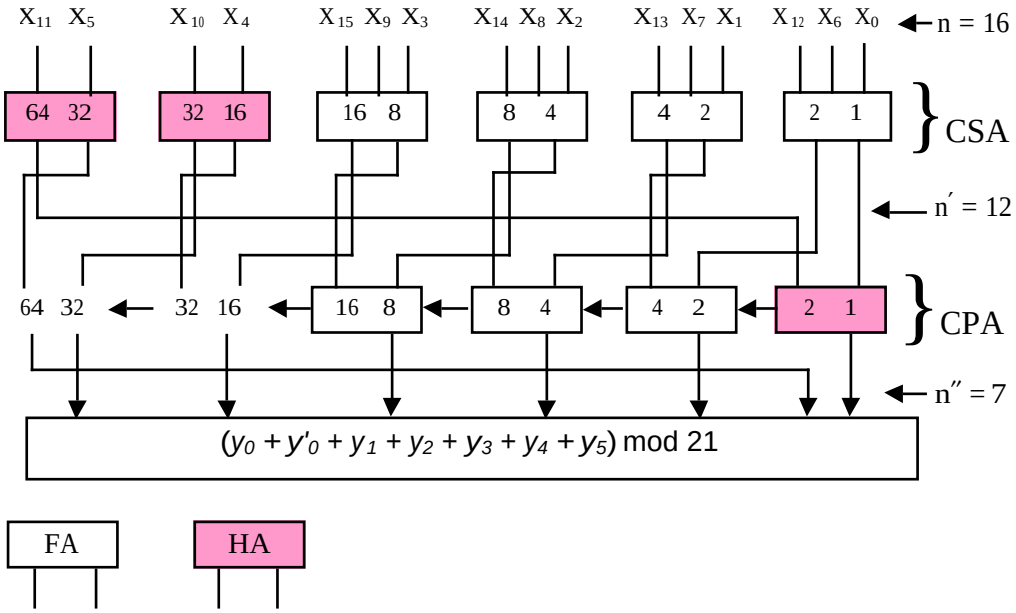


Figure 2.4. Générateur modulo 21 à 16 bits.

Les générateurs modulo les plus compacts sont obtenus pour les bases de la forme :
$A = 2^a - 1$ $(P(2^a - 1) = a)$. Dans ce cas le résidu est obtenu directement sans avoir besoin d'un bloc logique à la fin. La figure 2.5 représente un exemple de générateur modulo 7 à 16 bits.

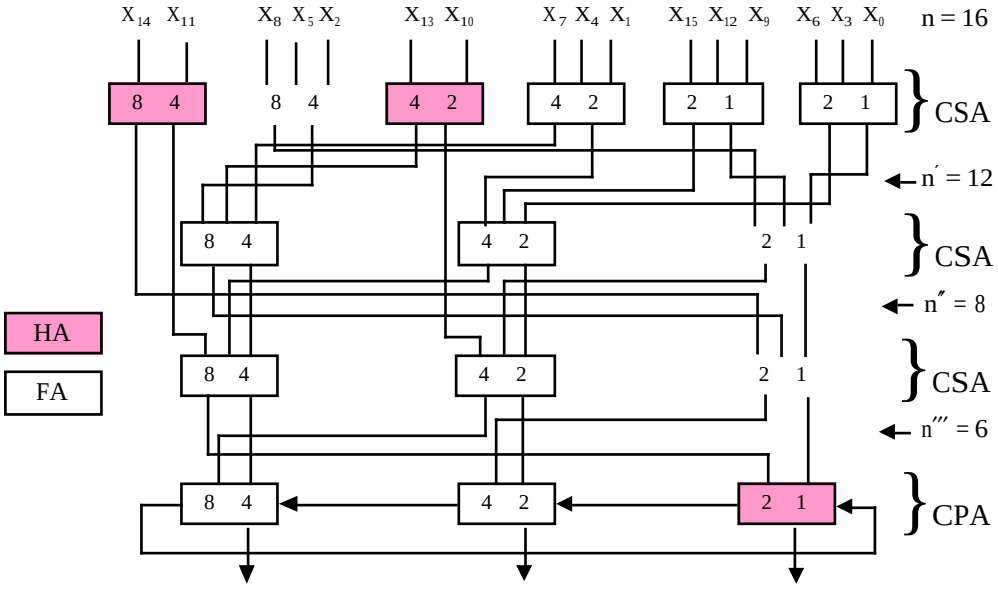


Figure 2.5. Générateur modulo 7 à 16 bits

2.2.2 Générateur Modulo A Utilisant HP(A)

Dans ce cas *per(A) = HP(A)*. Le générateur modulo est réalisé en utilisant les mêmes techniques que dans le cas du générateur basé sur *P(A)*. Les *n* bits d'entrée sont divisés en $v = \lceil n/HP(A) \rceil$ octets de *HP(A)* bits chacun (le dernier B_{v-1} peut être incomplet). Soit b_i la valeur décimale de B_i, (1) peut alors s'écrire :

$$|X|_A = \left| \sum_{i=0}^{v-1} b_i \cdot 2^{i \bmod HP(A)} \right|_A \quad (4)$$

En tenant compte de l'égalité $\left|2^{t \cdot HP(A)+j}\right|_A = (-1)^t \left|2^j\right|_A$ (5), (1) devient :

$$|X|_A = \left| \sum_{i=0}^{v-1} (-1)^j b_i \right|_A \quad (6)$$

En écrivant $\left|-b_i\right|_A = \left|\left(2^{HP(A)} - 1 - b_i\right) + \left(1 - 2^{HP(A)}\right)\right|_A = \left|\overline{b_i} + 2\right|_A$ on trouve :

$$|X|_A = \left| \sum_{i=0,even}^{v-1} b_i + \sum_{i=0,odd}^{v-1} \overline{b_i} + 2[v/2] \right|_A \quad (7)$$

Cette équation peut être implémentée par le biais de *CSAs/CPA*. Les retenues des *FAs* et *HAs* de l'ensemble $G_{HP(A)-1}$ sont inversées et redirectionnées dans G_0. Ceci nécessite une correction de $-l$ (-1 par retenue inversée). La valeur de la correction totale est donc :

$COR(n,A) = |2\lfloor v/2 \rfloor + w - l|_A$. avec : w existe et $w = \sum_{i=d}^{HP(A)-1} 2^i$ seulement et seulement si v est pair et l'ensemble B_{v-1} est incomplet (car B_{v-1} a ses plus significatifs d bits (*0's*) qui deviennent des *1's* après que B_{v-1} est complémenté).

Exemple 2 : Conception d'un générateur modulo 9 à 16 bits.
Dans ce cas $HP(9) = 3$. Les bits d'entrée sont d'abord divisés en $v = 6$ octets $B_0 = \{x_2, x_1, x_0\}$, $B_1 = \{x_5, x_4, x_3\}$, ⋯, $B_5 = \{x_{15}\}$. Les bits de B_1, B_3, B_5 sont complémentés et ensuite attribués aux sous-ensembles G_0, G_1, G_2. A partir des bits de G_0, G_1, G_2 le générateur est réalisé de la même manière que le générateur basé sur P(A) mais en inversant les bits de poids 3. Le schéma de ce générateur est représenté par la figure 2.6. Pour la valeur corrective on a $v = 6$, $l = 4$ et $w = 2+4 = 6$. On trouve $COR(16,9) = |8|_9 \equiv -1$ qui est la valeur reportée sur la figure.

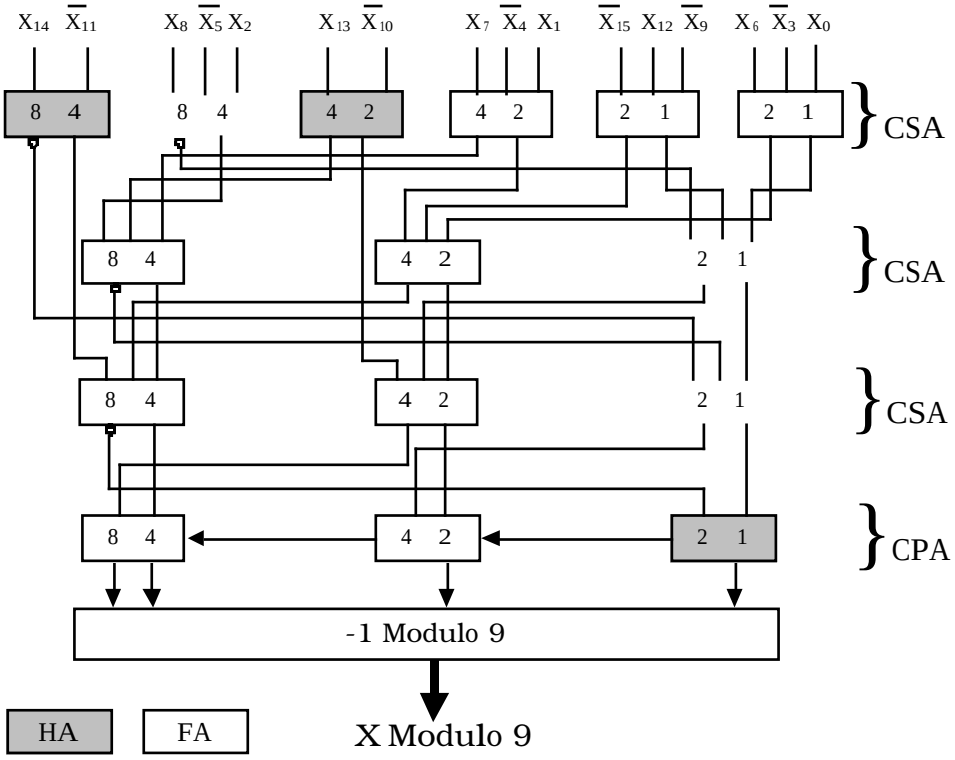


Figure 2.6. Générateur modulo 9 à 16 bits

2.2.3. Générateur Modulo A Utilisant Un MOMA

Les deux schémas précédents peuvent être inefficaces lorsque la période de *A* (ou la demi-période si elle existe) est grande par rapport à *n*. Dans ce cas, un additionneur de résidus (*MOMA*) modulo *A* peut être utilisé. Dans cette section on présente le schéma général d'un générateur de résidus et qui est basé sur les mêmes principes que les schémas précédents. Tout d'abord on va présenter le *MOMA*.

2.2.3.1. Additionneur Multi-Opérandes Modulo A (MOMA)

Soient :

$X_0 = (x_{a-1}, \_\_\_, x_1, x_0)$

$X_1 = (x_{2a-1}, \_\_\_, x_{a+1}, x_a), \_\_\_$, et

$X_{k-1} = (x_{ka-1}, \_\_\_, x_{(k-1)a})$

k résidus modulo *A.* Un Additionneur à *k opérandes* Modulo *A* (noté *(k, A) MOMA*) est le circuit qui calcule :

$$|X|_A = \left| \sum_{i=0}^{k-1} X_i \right|_A$$

qui est équivalent à

$$|X|_A = \left| \sum_{j=0}^{a-1} \left| 2^j \right|_A \left\lfloor \sum_{i=0}^{k-1} x_{ia+j} \right\rfloor_{2^{Per(A)}-1} \right|_A$$

Si on utilise les techniques précédemment présentées dans ce paragraphe le *MOMA* calcule d'abord la quantité :

$$\left| \sum_{j=0}^{a-1} \left| 2^j \right|_A \left\lfloor \sum_{i=0}^{k-1} x_{ia+j} \right\rfloor_{2^{Per(A)}-1} \right|_{2^{Per(A)}-1}$$

en utilisant un arbre de *CSA/CPA*, et ensuite il calcule le résultat final :

$$\left| \left| \sum_{j=0}^{a-1} \left| 2^j \right|_A \left\lfloor \sum_{i=0}^{k-1} x_{ia+j} \right\rfloor_{2^{Per(A)}-1} \right|_{2^{Per(A)}-1} \right|_A$$

Le nombre maximal d'ensembles G_j utilisés dans l'arbre *CSA/CPA* du *MOMA* est donné par
$q = \min \{Per(A), m\}$

Où : $m = \lceil log\ [k(A-1)+1] \rceil$ est le nombre de bits utilisé pour coder le nombre maximal résultant de l'addition de *k* résidus modulo *A* en binaire. Suivant la valeur de *A*, le *MOMA* est conçu selon les procédures suivantes :

Cas 1. $A \neq 2^{a-1}+1$

Procédure 2 :

Etape 1 : Calcul de $m = \lceil log\ [k(A-1)+1] \rceil$.

Etape 2 : Partition des $k \cdot a$ bits d'entrée en *a* sous-ensembles $G_j = \{x_{va+j} \mid 0 \le v \le k-1\}$ pour $0 \le j \le a-1$. On suppose que $G_j = \phi$ pour $a \le j \le q-1$.

Etape 3 : Réduire les $k \cdot a$ bits d'entrée en n' bits en utilisant des *CSAs* avec *EAC* de longueur allant à *q* jusqu'à ce qu'un seul sous-ensemble au plus a trois bits.

Etape 4 : Réduire n' en n'' en utilisant un *CPA* avec *EAC*; $n'' = q$ lorsqu'il n'y a pas de retenue sortante et $n'' = q+1$ sinon.

Etape 5 : Calcul du résultat final (le résidu) en utilisant un bloc logique ou une *ROM* et en tenant compte de la correction totale (si lieu).

Cas 2. $A=2^{a-1}+1$

Procédure 3 :

Dans ce cas $HP(A) = a\text{-}1 < a$ ($< m$) et le schéma du *MOMA* ressemble beaucoup au schéma du générateur modulo $2^{a-1}+1$. Les étapes de réalisation du générateur sont les suivantes :

Etape 1 : Partition des $k \cdot a$ bits d'entrée en $a\text{-}1$ sous-ensembles $G_0 = \{x_{ra}, \overline{x}_{ra+a-1} \mid 0 \le r \le k\text{-}1\}$, $G_j = \{x_{ra+j} \mid 0 \le r \le k\text{-}1\}$ pour $1 \le j \le a\text{-}2$. On suppose que $G_{a-1}=\phi$.

Etape 2 : Compression des $k \cdot a$ bits d'entrée en n' bits en utilisant des *CSAs* (avec des retenues rentrantes complémentées) de longueur allant à $a\text{-}1$ jusqu'à ce qu'un seul sous-ensemble (G_0 seulement) a trois bits.

Etape 3 : Compression des n' bits en a bits utilisant un additionneur séquentiel (*CPA*) à $a\text{-}1$ bits.

Etape 4 : Calcul de $\left| y_0 + y_1 + ??? + y_{a-1} + COR(k, 2^{a-1}+1) \right|_A$, où :

$COR(k, 2^{a-1}+1) = \left| -(l+k) \right|_{2^{a-1}+1}$.

Exemple 3 : Conception d'un générateur modulo 37 à 32 bits.

$Per(37) = HP(37) = 18$. Les bits d'entrée sont divisés en deux ensembles $B_0 = \{x_{17}, \_\_\_, x_0\}$ et $B_1 = \{x_{31}, \_\_\_, x_{18}\}$, et ensuite attribués aux sous-ensembles G_j. Dans ce cas, on ne peut pas profiter de la périodicité parce que $|G_j| = 2$ pour $0 \le j \le 13$ et $|G_j| = 1$ pour $14 \le j \le 17$ ($|G_j|$ est le cardinal de G_j). Le générateur peut être réalisé en utilisant un *MOMA* à 6 opérandes (ou (6, 37) *MOMA*). Les douze bits des ensembles G_j pour $0 \le j \le 5$ sont considérés comme deux opérandes. Les 20 bits restants peuvent être convertis en quatre résidus modulo 37 en utilisant quatre blocs logiques. Les 6 opérandes résultantes sont ensuite additionnées avec un (6, 37) *MOMA* conçu selon la procédure 2. La figure 2.7 représente le schéma de ce générateur.

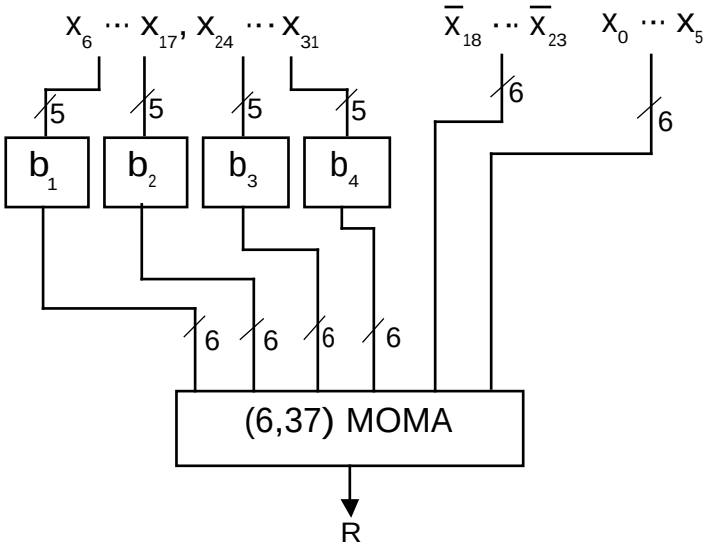


Figure 2.7. Schéma du générateur de résidus modulo 37 à 32 bits

Les blocs b_1, ___ b_4 convertissent chacun 5 bits d'entrée en un résidu modulo 37.

2.3. Le Contrôleur des Codes Résidu

Les codes à résidu sont des codes séparables. Chaque mot de code comporte des bits d'information (partie I) et des bits de contrôle (partie C). La structure générale d'un contrôleur pour codes séparables est présentée dans la figure 2.7. Dans cette figure, un générateur calcule

l'inverse des bits de contrôle (partie C^*) à partir des bits d'information (I). Ces résultats sont comparés aux bits de la partie C par le biais d'un contrôleur double-rail.

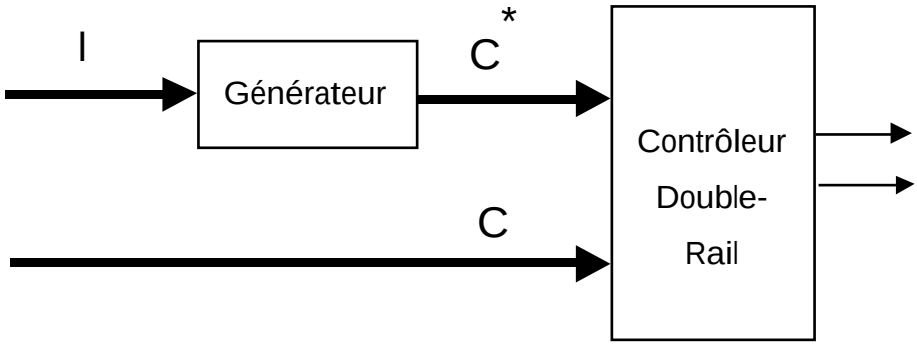


Figure 2.7. Contrôleur pour codes séparables

L'auto-testabilité du contrôleur vis à vis des fautes affectant le bloc générateur est assurée car les mots de code exercent exhaustivement cette partie. Néanmoins, dans certaines conditions le contrôleur double-rail peut ne pas recevoir un ensemble de 4 mots d'entrée testant toutes ses cellules. Dans ce cas, relativement rare, on peut soit mélanger les signaux C et C^* avec d'autres signaux double-rail disponibles dans le circuit, soit utiliser une paire de signaux double-rail générée par deux bascules [Nic94] afin d'assurer l'auto-testabilité de cette partie.

De manière générale, la structure de la figure 2.7 ne peut pas s'appliquer au cas des codes à résidu. La raison de cela se trouve dans la façon d'implémentation des générateurs de résidu qui peuvent produire deux représentations différentes de la valeur zéro. L'une étant la valeur 00 _ _ _ 0 et l'autre la valeur 11 _ _ _ 1. Par conséquent, l'utilisation de cette structure peut donner à certains instants des valeurs égales sur C et C^* (tous les deux à 00 _ _ _ 0, ou tous les deux à 11 _ _ _ 1), résultant en de fausses alarmes aux sorties du contrôleur double-rail. Pour résoudre ce problème, deux solutions sont possibles : la première est l'utilisation d'un circuit de translation, et la seconde est la suppression de l'une des deux valeurs du zéro en forçant le générateur de résidus à donner toujours une seule représentation. Ces solutions sont présentées par la suite.

Utilisation d'un Circuit de Translation

Dans ce cas on suppose que deux représentations du zéro sont possibles (00___ 0 et 11___1) à la sortie du générateur modulo *A.* Le contrôleur double-rail peut donc signaler une fausse alarme si on a ces deux représentations en même temps (C et C^* sont égales). Pour les bases de la forme 2^k-1, un circuit translateur permet de remédier à ce problème [Nik88]. Un exemple dans le cas de $A = 7$ est présenté dans la figure 2.8. De manière générale, on peut vérifier que pour :
A=B=11 _ _ _1, ainsi que pour A=B=00 _ _ _0, on obtient à la sortie :
A'=00 _ _ _0 et B'=11 _ _ _1.
Pour toutes les valeurs en dehors du code double-rail sur A et B on obtient des valeurs hors code double-rail sur A' et B'. D'autre part pour toutes les valeurs du code double-rail sur A et B, on obtient des valeurs du code double-rail sur A'et B'. Par conséquent, le translateur nous permet de résoudre le problème des deux représentations du zéro, tandis qu'en dehors de ces cas il n'affecte pas les mots dans le code et hors code.
On démontre que ce translateur est totalement auto-contrôlable avec les mots de code suivants : A = B = 0, A = 2^k-1-B, et A = B = 2^k-1.

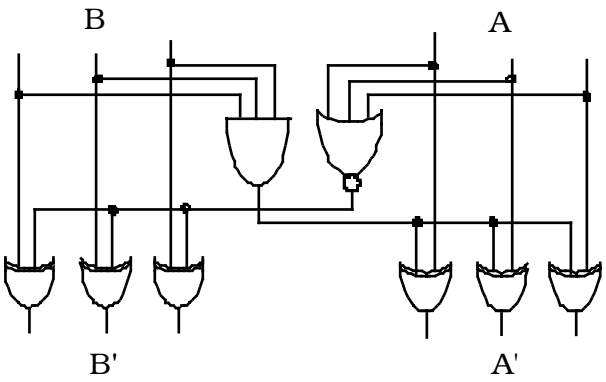


Figure 2.8. Exemple du translateur pour $A = 7$

Modification du Générateur de Résidus

Parmi les générateurs de résidus exposés dans les sections précédentes, les générateurs donnant deux représentations de la valeur zéro sont ceux qui utilisent un additionneur séquentiel à retenue rebouclée en dernière étape (bases de la forme 2^k-1). Pour éliminer la double représentation du zéro en modifiant la structure du générateur, plusieurs solutions existent dans la littérature. Une première solution est obtenue en réalisant un OU logique de la retenue sortante et des signaux de propagation de l'additionneur (voir chapitre 3). Le résultat est ensuite injecté comme retenue entrante [She77]. Cette solution élimine la représentation 11 _ _ _ 1 du zéro (elle élimine aussi l'instabilité de l'additionneur).

Une autre solution pour éliminer la double représentation du zéro consiste à remplacer de l'additionneur séquentiel par un additionneur à calcul anticipé des retenues (voir chapitre 3) [Efs94]. En remarquant que la retenue sortante dépend seulement des valeurs des signaux de propagation et de génération de l'additionneur, les équations donnant les retenues de l'additionneur sont recalculées en considérant la retenue sortante comme retenue entrante (rebouclage). Les équations résultantes permettent de réaliser deux implémentations régulières de l'additionneur. L'une donnant une seule représentation du zéro, l'autre permettant de garder la double représentation. Cette solution est bien évidemment plus coûteuse que celle de l'additionneur séquentiel, notamment pour les grandes bases. Un exemple d'une implémentation donnant une seule représentation du zéro pour A=7 est donné en figure 2.9. Pour raison de clarté, les interconnexions ne sont pas représentées sur la figure.

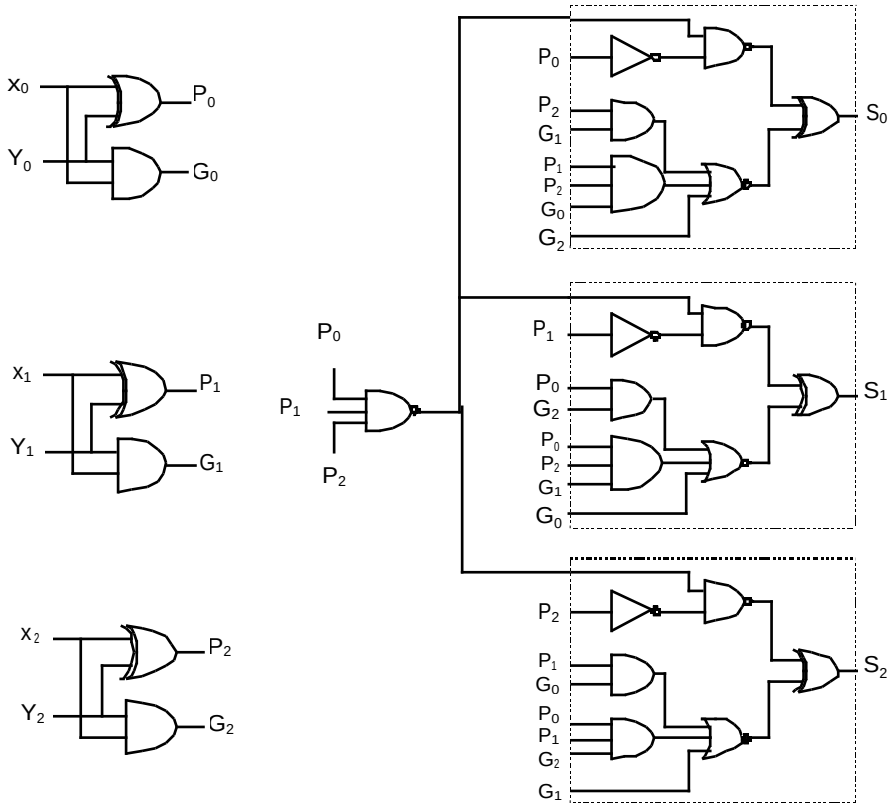


Figure 2.9. Additionneur modulo 7 avec une seule représentation du zéro

2.4. Outils de Génération

Nous avons intégré dans notre outil informatique des générateurs de macro-blocs paramétrables qui produisent, selon les procédures précédentes, des générateurs de résidus pour n'importe quelle base impaire. La figure 2.9 décrit l'algorithme réalisé.
Pour réaliser un bloc logique calculant un résidu, nous générons automatiquement un fichier contenant une description exhaustive du fonctionnement de ce bloc à partir du nombre de bits d'entrée, du poids de chacun de ces bits, et de la valeur de la base. Ce fichier est ensuite synthétisé pour produire le bloc décrit.

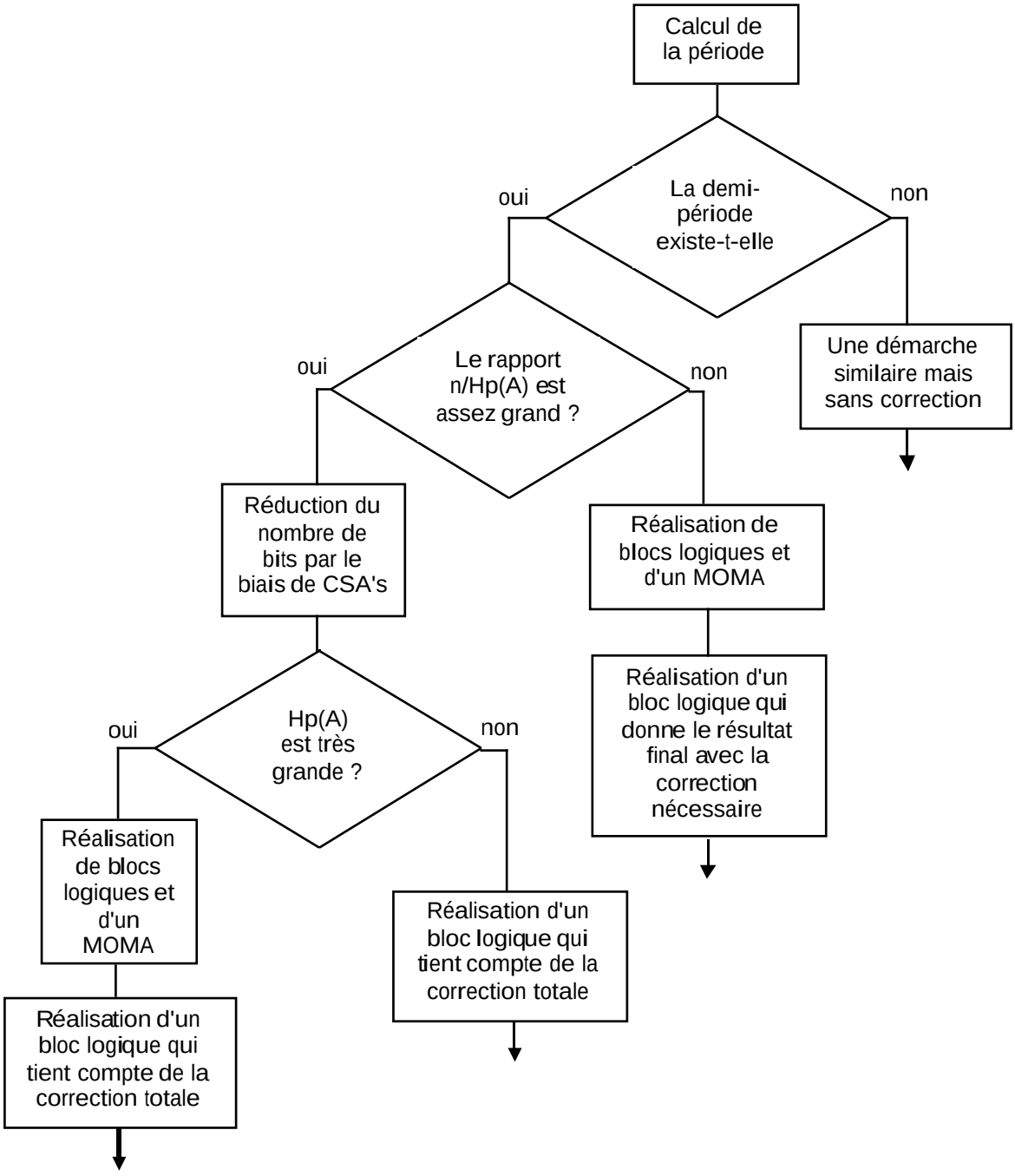


Figure 2.9. L'algorithme de réalisation d'un générateur de résidus

Le tableau 2.1 montre les résultats de simulations obtenus pour quelques bases qui seront utilisées plus tard. La technologie utilisée est la technologie ES2-CMOS-1µm.

Valeur de la base	16 bits		32		64		128	
	S (mm^2)	D (ns)	S (mm^2)	D (ns)	S (mm^2)	D (ns)	S (mm^2)	D (ns)
3	0,060	22,7	0,128	31,1	0,255	39,8	0,518	48,6
7	0,056	19,6	0,126	27,7	0,253	36,3	0,516	44,9
2^6-1	0,044	14,7	0,112	23,8	0,252	32,4	0,506	40,7
2^8-1	0,032	14,9	0,095	23,3	0,240	31,2	0,511	39,5
2^6+1	0,058	21,5	0,135	25,9	0,279	33,9	0,550	45,9
2^{12}-1	0,033	17,8	0,088	23,2	0,224	32,9	0,504	41,2
37	0,257	35,2	0,446	44,8	0,543	47,7	0,828	56,9

Tableau 2.1. Coût et performances de différents générateurs de résidus.

Chapitre 3. Additionneurs Auto-Contrôlables Basés sur le Code Résidu

La conception d'additionneurs rapides est d'une grande importance, non seulement pour effectuer des additions, mais également pour construire des multiplieurs et des diviseurs efficaces. Dans cette partie on va présenter trois types d'additionneurs qui sont parmi les plus utilisés en pratique. Pour chacun de ces additionneurs, on fait une analyse en présence de fautes cellulaires simples. Cette analyse permet de déterminer les valeurs arithmétiques possibles des erreurs générées par des fautes simples et ensuite de calculer la base du code de résidu garantissant la sûreté en présence de fautes.

3.1. Rappels

Soit $A = A_{n-1}\ A_{n-2}\_\_\_A_0$ et $B = B_{n-1}\ B_{n-2}\_\_\_B_0$ deux nombres écrits en base 2. Le résultat de l'addition de A et B est $S = S_n\ S_{n-1}\ S_{n-2}\_\_\_S_0$ avec :

$S_i = A_i \oplus B_i \oplus C_i$ pour $i < n$

$S_n = C_n$

$C_{i+1} = C_i\_(A_i + B_i) + A_i\_B_i$

C_i est la retenue de poids i et le terme S_n indique le dépassement. A partir des équations précédentes on peut construire une cellule de base appelée *FA* (*Full Adder*) qui calcule S_i et C_{i+1} en fonction de A_i, B_i *et* C_i.

3.2. Additionneur Séquentiel (Ripple Carry Adder)

L'additionneur séquentiel est l'additionneur le plus simple. Il est composé de cellules de *FA* connectées en série. La figure 3.1 représente un exemple d'additionneur séquentiel sur 4 bits.

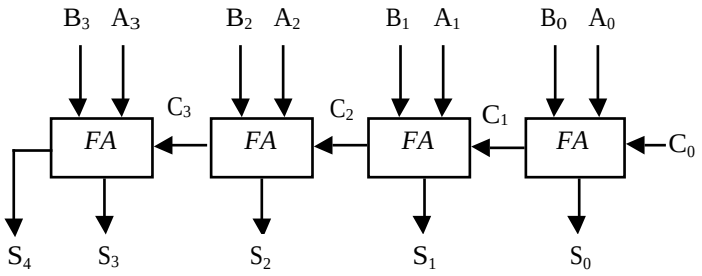


Figure 3.1. Additionneur séquentiel de 4 bits

3.2.1. Analyse en Présence de Fautes et Calcul de la Base

Une faute dans une cellule de FA produit toujours une erreur simple si la partie qui génère la somme est indépendante de la partie qui génère la retenue comme dans le cas de la figure 3.2. Une erreur simple sur l'une des sorties C ou S a une valeur arithmétique qui prend la forme $\pm 2^i$ (erreur arithmétique simple). Une telle erreur n'est pas divisible par 3. On peut alors la détecter en utilisant la base A=3.

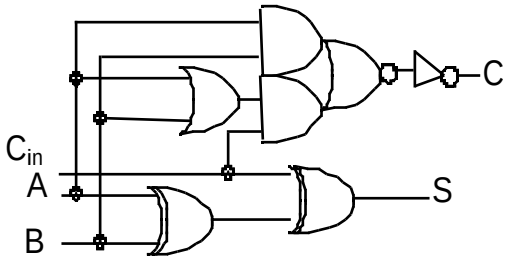


Figure 3.2. Une cellule de FA sans partage de portes logiques entre les sorties

Lorsque les sorties *S* et *C* partagent des portes logiques, les deux signaux *S* et *C* peuvent être erronés en même temps. Ceci est le cas de la figure 3.3. Dans ce cas l'erreur peut prendre en plus des valeurs $\pm 2^i$ les valeurs $\pm 3\_2^i$. Puisque 3_2^i est divisible par 3 il faut utiliser la valeur 7 comme base de détection d'erreur pour atteindre la sûreté en présence de fautes dans ce cas. Néanmoins, dans la figure 3.3, une erreur double sur les deux signaux de sortie *S* et *C* arrive lorsque l'erreur passe par le signal commun *P*. Dans ce cas, l'erreur se propage toujours vers *S* ($S = P \oplus Cin$). D'autre part, on a $C = P\_Cin + A\_B$, et alors l'erreur sur *P* se propage vers *C* si et seulement si $Cin = 1$ et $A\_B = 0$. Dans ce cas, on a : $S = \overline{P}$ et $C = P$. La valeur arithmétique de cette erreur double est $2^{i+1} - 2^i = 2^i$ pour $P = 0 \rightarrow 1$ et $-2^{i+1} + 2^i = -2^i$ pour $P = 1 \rightarrow 0$. Dans les deux cas, l'erreur est simple et par conséquent, la sûreté en présence de fautes sera assurée en utilisant aussi la valeur 3 comme base de détection d'erreur, identique au cas de la figure 3.2.

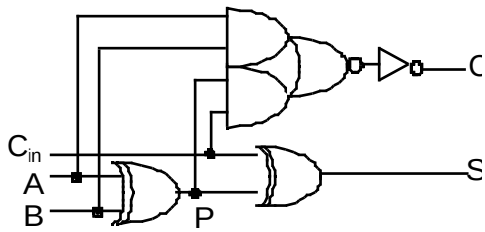


Figure 3.3. Une cellule de FA avec partage de portes logiques entre les sorties

Si une faute affecte l'une des cellules FA de l'additionneur séquentiel, l'erreur générée sera propagée dans les cellules FA suivantes et le résultat final sera modifié de la même valeur que l'erreur d'origine. La base 3 permet donc la détection des erreurs produites par n'importe quelle faute cellulaire simple si l'une des cellules FA présentés ci-dessus est utilisée pour réaliser l'additionneur.

3.3. Additionneurs à retenue anticipée (Carry Lookahead Adder)

Ces additionneurs cherchent à calculer à l'avance les valeurs des retenues grâce à des circuits logiques supplémentaires [Hwa79] [Mul94] [Wes94]. Si l'on additionne deux nombres $A = A_{n-1}\ A_{n-2}\_\_\_A_0$ et $B = B_{n-1}\ B_{n-2}\_\_\_B_0$ écrits en base 2, alors le comportement de la retenue est entièrement déterminé par les quantités :

$P_i = A_i \oplus B_i$ (si $P_i = 1$, la retenue précédente est propagée par l'étage *i*)

$G_i = A_i . B_i$ (si $G_i = 1$, l'étage *i* génère une retenue)

On peut remarquer que l'étage i de l'additionneur reçoit une retenue entrante C_i égale à 1 si et seulement si :

- L'étage *i-1* a généré une retenue ($C_{i-1} = 1$)

ou - L'étage *i-1* a propagé une retenue générée à l'étage i-2 ($P_{i-1} = G_{i-2} = 1$)

ou - Les étages *i-1* et *i-2* ont propagé une retenue générée par l'étage *i-3* ($P_{i-1} = P_{i-2} = G_{i-3} = 1$)

___

ou - Les étages *i-1, i-2, i-3, ___, 1, 0* ont propagé une retenue entrant dans l'additionneur ($P_{i-1} = P_{i-2} = \_\_\_ = P_1 = P_0 = C_0 = 1$).

Cette remarque nous donne le résultat fondamental suivant, qui peut d'ailleurs se démontrer par récurrence en utilisant le fait que C_i est égal à $G_{i-1} + C_{i-1}.P_{i-1}$:

$C_i = G_{i-1} + G_{i-2}.P_{i-1} + G_{i-3}.P_{i-2}.P_{i-1} + \_\_\_ + G_0.P_1.P_2 \_\_\_ P_{i-1} + C_0.P_0.P_1 \_\_\_ P_{i-1}$.

Le principe des additionneurs à évaluation anticipée de la retenue, appelés en anglais *Carry Look Ahead* (*CLA*), consiste à évaluer de manière parallèle, en trois étapes :

- les couples (G_i, P_i)
- les retenues C_i à l'aide de la formule ci-dessus
- les bits de la somme $S_i = A_i \oplus B_i \oplus C_i = P_i \oplus C_i$

La figure 3.4 représente le schéma d'un CLA.

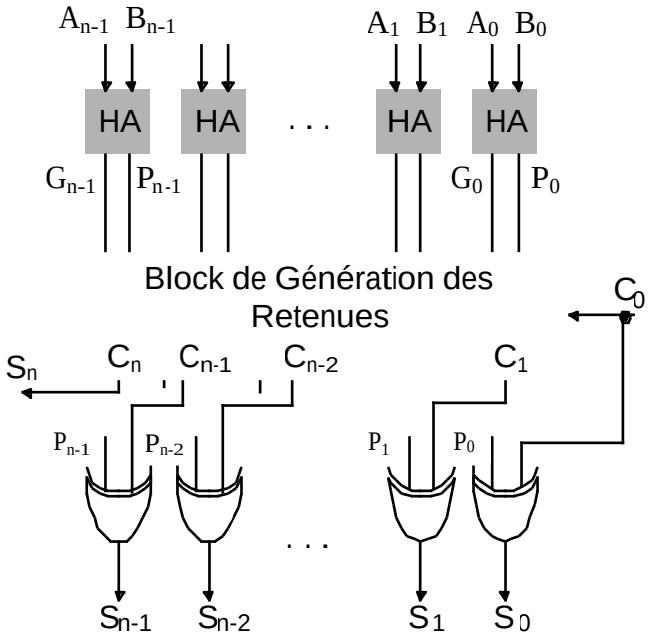


Figure 3.4. Schéma d'un CLA

La formule qui donne les retenues est très lourde, et par conséquent ce principe ne pourra s'appliquer qu'à de petits additionneurs (4 à 8 bits). Pour cette raison, les additionneurs de type CLA sont utilisés comme petits blocs constitutifs de plus gros additionneurs.

Dans notre cas, le bloc de génération des retenues isé en le découpant en blocs de quatre bits au maximum (Carry Lookahead Unit ou CLU). La figure 3.5 représente le schéma logique d'un bloc de 4 bits. P^* caractérise l'aptitude du bloc de 4 bits à propager la retenue entrante C_0. De même, G^* caractérise l'aptitude de ce bloc à générer une retenue sortante C_4 (C_4 est égale à $G^* + P^*\_C_0$).

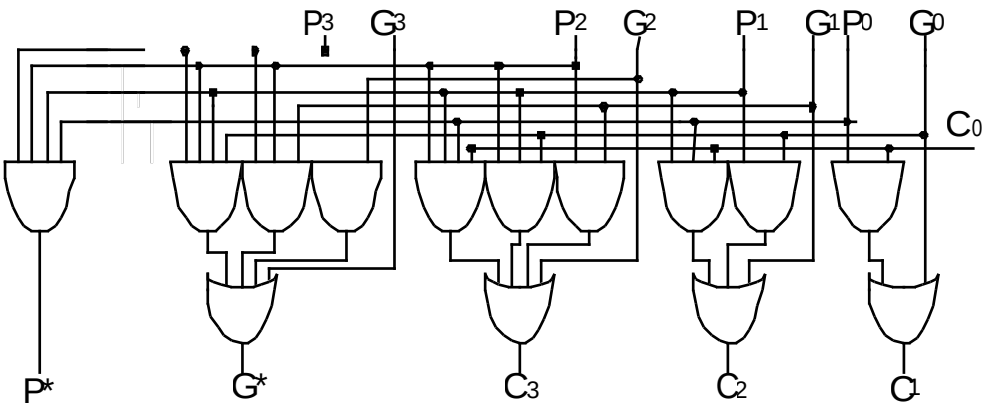


Figure 3.5. Une CLU 4 bits élément d'un CLA cascadé

La disposition de ces blocs de base selon un arbre permet de réaliser des additionneurs de n'importe quelle taille. La figure 3.6 présente le schéma symbolique d'un bloc de génération de retenues à 16 bits ainsi réalisé. Ces additionneurs sont appelés des CLA cascadés.

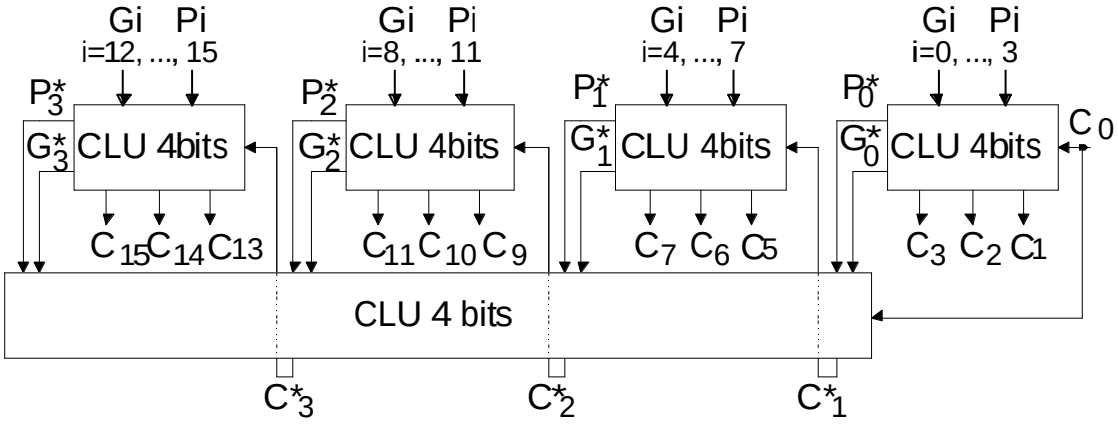


Figure 3.6. Un CLA cascadé 16 bits

3.3.1. Analyse en Présence de Fautes et calcul de Base

Cette analyse est divisée en trois parties selon l'emplacement de la faute dans le circuit de l'additionneur (voir figure 3.4).

Si c'est l'une des portes logiques XOR donnant une sortie primaire qui est affectée, il est évident que la valeur arithmétique de l'erreur est de forme $\pm 2^i$ où i est le poids du signal erroné.

Si maintenant la faute se trouve dans le bloc de HAs, on aura un seul signal P_i ou G_i erroné. Si c'est le signal G_i qui est erroné, la valeur arithmétique de l'erreur est de la forme $\pm 2^i$ puisque G_i est une retenue. La valeur arithmétique de cette erreur s'additionne au résultat de l'additionneur pour donner une erreur de la même forme. Supposons maintenant que c'est le signal P_i qui est erroné. On sait que tous les termes de propagation d'indice $j > i$ sont dépendant de ce signal. Soit $k > i$ tel que P_k soit le premier des P_j dont la valeur est à 0. L'erreur sur P_i sera répercutée sur les termes de propagation d'indices ($i+1$) à ($k-1$) car les signaux P_{i+1} ___ P_{k-1} ont la valeur 1. On aura dans ce cas une retenue propagée incorrectement sur les sorties i ___ ($k-1$), mais la valeur arithmétique de l'erreur est égale à la valeur de cette retenue.

Le troisième cas de figure est celui d'une faute dans le bloc de génération des retenues. La faute va se trouver dans l'un des éléments de base du bloc à savoir une CLU de 4 bits. Les signaux de sortie d'un élément de base ne partagent pas de portes logiques et par conséquent

une seule sortie sera erronée. Le signal erroné sera donc soit une retenue soit la propagation ou la génération d'une retenue (P^* ou G^*). La valeur arithmétique de l'erreur est également de la forme $\pm 2^i$ si c'est une retenue ou la génération de retenue qui est affectée. Si l'erreur se trouve sur le signal de propagation, la même analyse faite plus haut s'applique.

On peut donc conclure que dans tous les cas de figure, une faute cellulaire simple dans le circuit de l'additionneur CLA présenté auparavant génère une erreur de valeur arithmétique simple détectable par le code arithmétique de base 3.

3.4. Additionneurs Rapides Utilisant la Cellule de Brent & Kung

Les additionneurs présentés dans cette section utilisent les mêmes principes que les CLA, mais sont plus réalistes pour une taille raisonnable des opérandes, et plus simples que les CLU cascadés. La figure 3.7 représente la structure générale de ces additionneurs. Ils diffèrent les uns des autres par la structure du bloc de calcul des retenues (bloc en gras). Ces blocs utilisent tous une cellule de base que l'on va appeler la cellule de Brent et Kung. La figure 3.8 représente le schéma logique de la cellule de Brent et Kung.

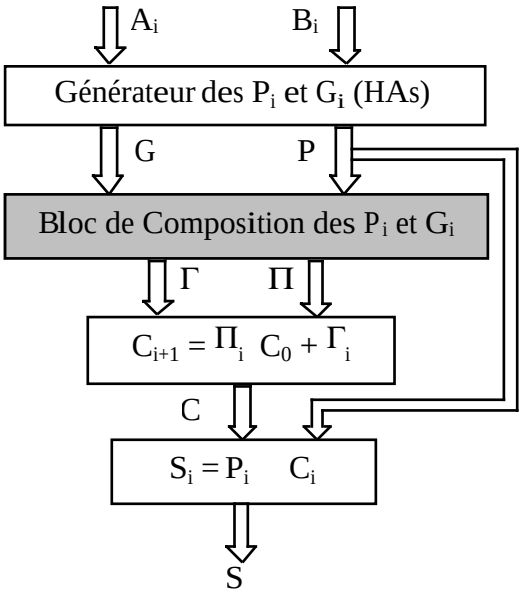


Figure 3.7. Structure générale des additionneurs utilisant la cellule de Brent et Kung

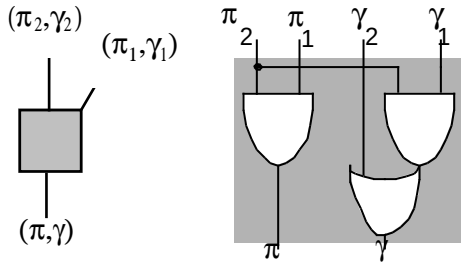


Figure 3.8. Schéma de la cellule de Brent et Kung

Le bloc de composition des quantités P_i et G_i est constitué de *n* arbres (*n* est la taille des opérandes) en parallèle qui évaluent les fonctions (P_j,G_j) o (P_{j-1},G_{j-1}) o ___ (P_0,G_0) pour $j \in [0, n-1]$. o représente l'opérateur de composition défini par :

(π_2,γ_2) o $(\pi_1,\gamma_1) = (\pi,\gamma) = (\pi_2 \_ \pi_1, \pi_2 \_ \gamma_1 + \gamma_2)$

Grâce à la structure en arbres parallèles, le bloc de composition a un délai proportionnel au logarithme de la taille des opérandes. Pour des raisons d'économie, les arbres sont superposés (les cellules qui exécutent les mêmes opérations dans différents arbres sont communes entre ces arbres).

La première structure d'additionneur de ce type a été présentée par Sklansky en 1960 [Skl60]. La figure 3.9 représente un exemple du bloc de calcul des retenues pour 16 bits. Les cellules grises et blanches sont des cellules de Brent&Kung. Les cellules grises prennent des entrées inversées et produisent des sorties normales et les cellules blanches prennent des entrées normales pour générer des sorties inversées. L'inconvénient majeur de ce schéma est la sortance importante qui augmente avec la taille de l'additionneur.

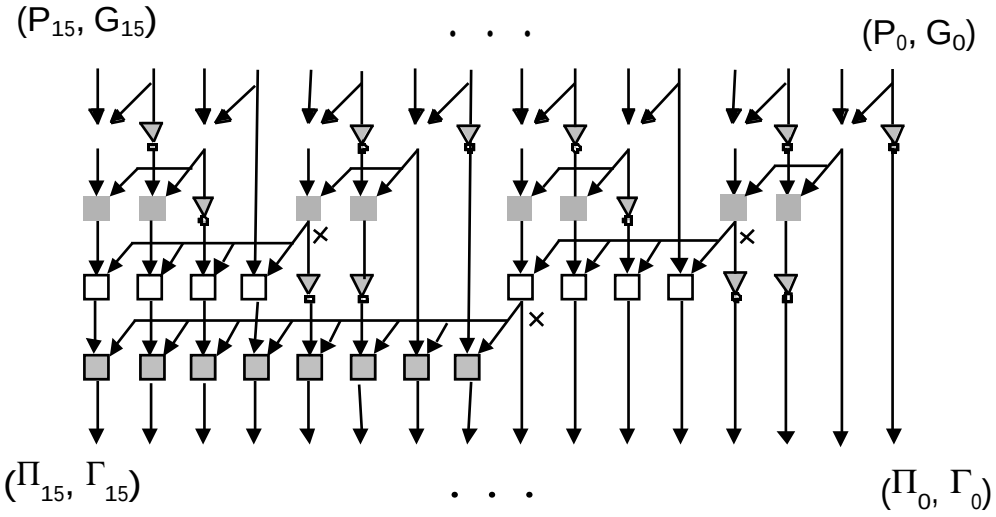


Figure 3.9. Sklanski 16 bits

Plus tard, Kogge et Stone ont proposé une autre structure reposant sur le même principe [Kog73]. Cette structure, présentée dans la figure 3.10, est la plus rapide de toutes les structures développées dans cette section. Elle a une sortance constante, mais aussi un coût en surface supérieur par rapport aux autres structures.

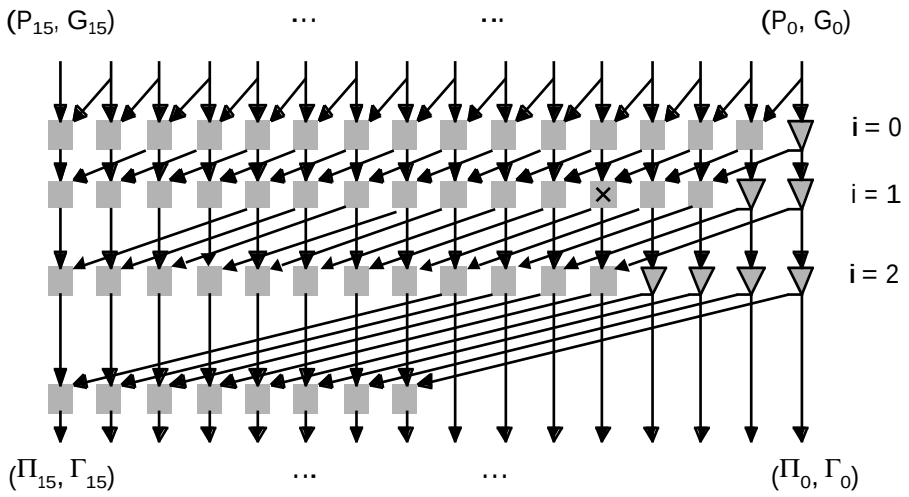


Figure 3.10. Structure de Kogge et Stone pour 16 bits

En 1982, dans [Bre82], R.P. Brent et H.T. Kung ont présenté la structure de coût minimum que l'on peut avoir en utilisant ce principe de composition. Cette structure a une sortance supérieure à celle de Kogge et Stone. La figure 3.11 présente un exemple de cette structure pour 16 bits.

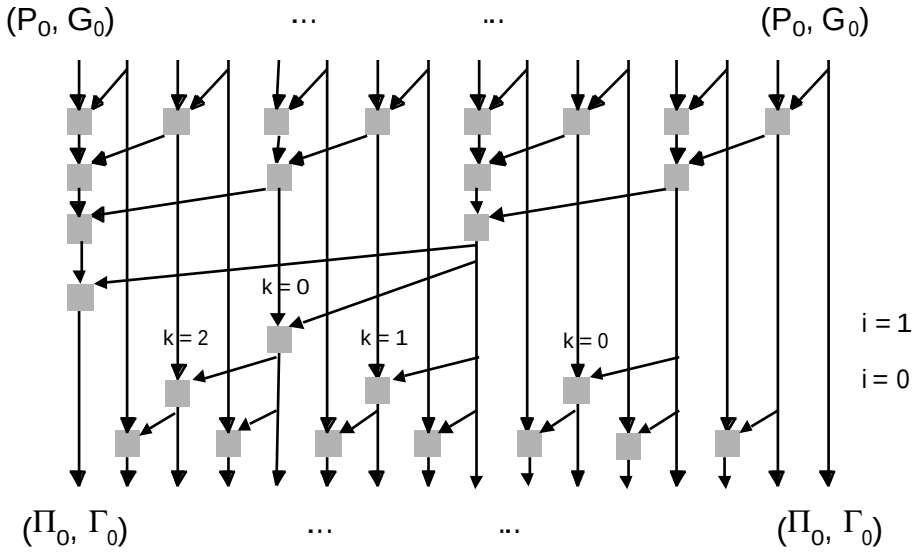


Figure 3.11. Structure de Brent et Kung pour 16 bits

Han et Carlson ont présenté leur structure en 1987 [Han87]. Cette structure est présentée par la figure 3.12. En considérant les facteurs vitesse et surface, la structure de Han et Carlson se situe entre les deux structures précédentes.

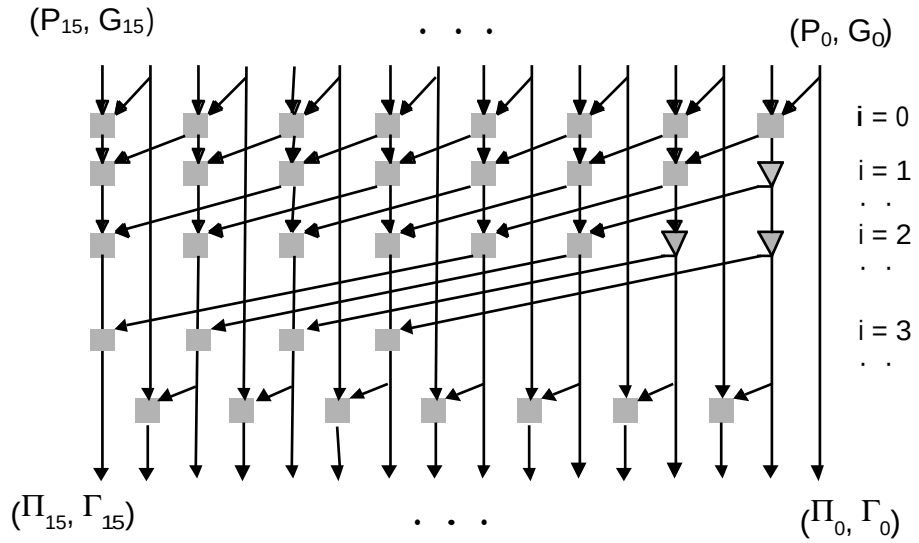


Figure 3.12. Structure de Han et Carlson pour 16 bits

Une comparaison générale entre les différentes structures d'additionneurs présentées ci-dessus est reportée dans le tableau 3.1.

Type	# de cellules	Performance	Max. Sortance	ex. 32-bits.		
Sklanski	$\lceil n/2 \log_2(n) \rceil$	$\lceil \log_2(n) \rceil$	n/2	80	5	16
Kogge-Stone	$\lceil n (\log_2 n - 1) + 1 \rceil$	$\lceil \log_2 n \rceil$	constant=2	129	5	2
Brent-Kung	$\lceil 2n - \log_2 n - 2 \rceil$	$\lceil 2 \log_2 n - 2 \rceil$	$\lceil 2 \log_2 n - 2 \rceil$	57	8	5
Han-Carlson	$\lceil n/2 \log_2 n \rceil$	$\lceil \log_2 n + 1 \rceil$	constant=2	80	6	2

Tableau 3.1. Comparaison entre les différentes structures d'additionneurs à base de cellule de Brent&Kung.

3.4.1. Analyse en Présence de Fautes

L'analyse en présence de fautes et la sélection de la base de détection d'erreur pour ce type d'additionneurs ont été étudiées dans [Spa93]. Le problème dans ce cas est le suivant : les retenues sont générées par un bloc de calcul anticipé qui a une structure totalement différente d'une structure basée sur la cellule d'addition de base (FA). La propagation d'une erreur est donc différente et peut produire des erreurs arithmétiques qui ne sont pas simples.

Dans la suite on utilisera les notions de dépendance fonctionnelle et de dépendance structurelle d'un signal C_j d'un signal W_i. Le terme de dépendance fonctionnelle est ici utilisé en différenciation d'une dépendance structurelle. La dépendance structurelle signifie que dans

l'implémentation du circuit il y un chemin menant de W_i à C_j. La dépendance fonctionnelle signifie que la valeur de C_j dépend de la valeur de W_i, mais il se peut que dans l'implémentation du circuit il n'y ait pas de chemin menant de W_i à C_j. Ceci peut survenir pour des raisons d'optimisation. Par exemple la cellule générant le signal W_i peut être dupliquée pour réduire sa sortance.

Pour déterminer les valeurs arithmétiques des erreurs à la sortie de l'additionneur, on considère les deux points suivants :

1. Seule une faute sur un signal dont la divergence est > 1 peut produire des erreurs différentes de $\pm 2^i$.

2. Soit une erreur sur un signal W_i de divergence > 1. Cette erreur peut se propager vers toutes les retenues C_j $(j \geq i)$ qui sont structurellement dépendantes de ce signal. On peut distinguer **deux cas différents** :

   cas 1 : toutes les retenues C_j $(j \geq i)$ sont dépendantes de W_i (c'est le cas du CLA).

   cas 2 : seulement une partie des retenues C_j $(j \geq i)$ est dépendante de W_i (les autres sont fonctionnellement dépendantes mais pas structurellement).

   Dans le premier cas, une faute sur W_i produit toujours une erreur de la forme :

   $$e = \pm(2^{i+k} - 2^{i+k-1} - 2^{i+k-2} - \ldots - 2^{i+1}) = \pm 2^{i+1} \quad (1)$$

   et qui est une erreur simple. Dans le deuxième cas, les erreurs peuvent être obtenues du premier cas $[\pm(2^{i+k} - 2^{i+k-1} - 2^{i+k-2} - \ldots - 2^{i+1})]$ en enlevant les termes qui ne sont pas structurellement dépendants de W_i. La forme générale de ces erreurs est donc la suivante :

   $$e = \pm[a_0(2^{i0+1} + 2^{i0+2} + \ldots \pm 2^{i0+k0}) + a_1(2^{i1+1} + 2^{i1+2} + \ldots \pm 2^{i1+k1}) + \_\_\_ +$$

   $$a_m(2^{im+1} + 2^{im+2} + \ldots \pm 2^{im+km})] \quad (2)$$

   où $a_0, a_1, \_\_\_ a_m \in \{0,1\}$, $i_{l+1} > i_l + r_l$, $k_l \in \{1, 2, \ldots, r_l\}$ et $i_m + r_m < n$.

A la formule précédente s'ajoutent les conditions suivantes :

- a_{q+1}= 0 si a_q=0. La propagation de l'erreur est stoppée à un stade intérieur.

- a_{q+1}= 0 si le signe de 2^{iq+kq} est négatif. Dans ce cas $P_{iq+kq+1}$ = 0 et la propagation de l'erreur est stoppée au niveau $iq+kq+1$. Les sorties de poids > $iq+kq$ sont correctes.

- le signe de 2^{iq+kq} est toujours négatif si k_q < r_q. C'est P_{iq+kq}= 0 qui stoppe la propagation de l'erreur.

Exemple : prenons le cas de l'additionneur de Kogge & Stone (figure 3.10). Soit une faute sur la cellule marquée par une croix à la ligne i=1. Les valeurs d'erreurs possibles pour cette faute sont :

$$\pm 2^5, \pm(2^9-2^5), \pm(2^9+2^5), \pm(2^{13}-2^9-2^5), \pm(2^{13}+2^9+2^5).$$

L'application de l'analyse précédente sur chacune des structures étudiées permet de trouver les formules donnant les valeurs d'erreurs possibles pour chaque cas. Ces formules sont en fait des sommes de séries géométriques. Dans le cas de la structure de Sklansky le **cas 1** s'applique et l'erreur est toujours de la forme $\pm 2^{i+1}$ (détectable avec la base 3).

En adoptant les notations suivantes, on trouve les formules données ci-dessous pour les autres structures :

n : nombre de bits de l'additionneur.
i : le numéro de ligne de la faute (voir figures).
d : est le numéro de colonne de la faute (voir figures).
les C_k : les valeurs d'erreurs possibles.

a) Kogge et Stone:

$0 \le i < \log_2(n) - 1,\ 0 \le d < n$

$$C_1 = 2^{d+1} \cdot \frac{2^{(k+1)\cdot 2^{i+1}} - 1}{2^{2^{i+1}} - 1},\ k \cdot 2^{i+1} + d + 1 < n$$

$$C_2 = 2^{d+1} \cdot \left(2^{k\cdot 2^{i+1}} - \frac{2^{k\cdot 2^{i+1}} - 1}{2^{2^{i+1}} - 1}\right),\ k \cdot 2^{i+1} + d + 1 \le n$$

b) Brent et Kung :

$$C = 2 \cdot \left(2^{2^{i+1}} - 1\right) \cdot 2^{k \cdot 2^{i+2} + 3 \cdot 2^{i+1} - 1}$$

$$0 \le i < \log_2(n) - 2$$

$$k \cdot 2^{i+2} + 3 \cdot 2^{i+1} - 1 < n$$

c) Han et Carlson :

$$1 \le i \le \log_2(n) - 1,\ 0 \le d < \frac{n-2}{2}$$

$$C_1 = 12 \cdot 2^{2d} \cdot \frac{2^{(k+1)\cdot 2^{i+1}} - 1}{2^{2^{i+1}} - 1},\ k \cdot 2^{i+1} + 2d + 2 < n$$

$$C_2 = 2^{2d+1} \cdot \overline{2^{k \cdot 2^{i+1}}} - 6 \cdot \frac{2^{k \cdot 2^{i+1}} - 1}{2^{2^{i+1}} - 1},\ k \cdot 2^{i+1} + 2d + 2 \le n$$

3.4.2. Calcul de la Base

L'ensemble des erreurs obtenues pour une structure donnée d'additionneur est ensuite utilisé pour déterminer la base du code à résidu permettant d'assurer la sûreté en présence de fautes. Pour garantir cette propriété, la base ne doit pas diviser aucune des valeurs d'erreur possibles pour l'additionneur analysé. L'algorithme qui recherche la base est simple. Il commence avec la base 3 et vérifie si aucune des erreurs n'est divisible par 3. Si c'est le cas il s'arrête, sinon il répète l'opération jusqu'à trouver le premier nombre impair qui ne divise aucune des valeurs d'erreur possibles. Si ce nombre est de la forme 2^k-1 (base à faible coût) la recherche s'arrête. On utilisera cette base pour implémenter le circuit. Si le nombre trouvé est de la forme 2^k+1 on le retient comme une base possible, mais on continue jusqu'à trouver un nombre du type 2^k-1 qui ne divise aucune des erreurs. Si on trouve un tel nombre avant d'atteindre la valeur 2^n-1 (n étant le nombre des bits d'informations), on la retient comme une seconde base possible. Ensuite, en utilisant les différents générateurs que nous avons développés, on implémente le circuit pour les deux bases et on retient celui ayant le moindre coût. Si la première base trouvée n'est ni de la forme 2^k-1 ni de la forme 2^k+1 on continue la recherche. Si la seconde base trouvée est de la forme 2^k-1 on évalue les deux solutions et on garde la moins coûteuse. Si elle est de forme 2^k+1, on la garde comme seconde base possible et on continue la recherche pour trouver une base du type 2^k-1. On évalue les différentes bases et on garde la moins coûteuse.

L'intégration de ces formules dans l'outil informatique permet de calculer les valeurs possibles de la base de détection d'erreurs (la base ne doit pas diviser les valeurs d'erreurs possibles). Les résultats trouvés sont reportés dans le tableau 3.2.

Les bases possibles	8 bits	16 bits	32 bits	64 bits	128 bits
Kogge&Stone	9, ___, 31, ___	23, ___, 63, ___	37, ___, $2^{11}+1$, ___ $2^{12}-1$	37, ___, $2^{11}+1$, ___ $2^{12}-1$	37, ___, $2^{11}+1$, ___ $2^{12}-1$
Brent&Kung	5, 7, ___	7, ___	7, ___	7, ___	7, ___
Han&Carlson	7, ___	15, ___	19, ___, 65, ___, 255	19, ___, 65, ___, 255	19, ___, 65, ___, 255

Tableau 3.2. Les valeurs de bases possibles pour les différentes structures d'additionneurs.

Dans le tableau 3.2 sont reportées pour chaque cas la première valeur trouvée et les valeurs de la forme 2^k+1 et 2^k-1. Ces dernières peuvent être intéressantes par rapport à une autre base même pour k élevée. Par exemple dans le cas de la structure de Kogge&Stone la première base trouvée pour $n \geq 27$ est 37. La valeur de la forme 2^k-1 existe pour k = 12. $2^{12}-1$ est beaucoup plus grande que 37, mais elle est plus économique (voir résultats chapitre 2).

3.5. Implémentation et Résultats

Nous avons implémenté des générateurs de macro blocs paramétrables qui génèrent les différents additionneurs étudiés. Le tableau 3.3 représente les performances et les dimensions trouvées pour différentes tailles en utilisant les cellules standards de la technologie ES2-CMOS-1 µm.

Type d'additionneur	8 bits		16		32		64	
	D (ns)	S (mm^2)	D	S	D	S	D	S
Kogge-Stone	8,56	0,0670	10,23	0,152	11,52	0,346	14,30	0,778
Han-Carlson	11,08	0,0627	12,89	0,141	14,70	0,313	16,96	0,691
Sklansky	10,67	0,0621	13,12	0,139	15,48	0,292	19,37	0,655
CLA	11,48	0,0614	11,70	0,127	16,88	0,255	17,96	0,514
Brent-Kung	11,50	0,0607	15,12	0,125	19,50	0,256	24,26	0,521

Tableau 3.3. Performances et coût des différents additionneurs.

Le tableau 3.4 montre le surcoût en % d'un additionneur CLA sûr en présence de fautes basé sur le schéma du code résidu. Bien que la base de détection d'erreur soit dans ce cas la base de coût minimum (base =3), le surcoût dépasse largement les 100% (plus que la duplication). Pour cette raison, le schéma d'un additionneur sûr en présence de fautes basé sur le code résidu n'est pas utilisé. L'utilisation de la parité s'avère beaucoup plus intéressante dans ce cas. Elle introduit un surcoût de 20% à 40% [Dua97c].

Dimensions (mm^2)	8 bits	16	32	64
CLA	0,0614	0,127	0,255	0,514
CLA (FS)	0,1716	0,350	0,704	1,385
Surcoût	179,5	175,6	176,1	169,5

Tableau 3.4. Dimensions et surcoût pour un additionneur CLA basé sur le code résidu

Chapitre 4. Multiplieurs Auto-Contrôlables Basés Sur le Code Résidu

Dans ce chapitre on va étudier l'implémentation sûre en présence de fautes de plusieurs structures de multiplieurs proposées dans la littérature et utilisées dans la pratique.
Le modèle de fautes considère des fautes affectant une seule porte logique du circuit à un moment donné. Une faute produit une erreur à la sortie de la porte logique. Cette erreur peut ensuite se propager jusqu'aux sorties primaires du multiplieur. Pour garantir la sûreté en présence de fautes, les effets d'une faute sur les sorties primaires doivent être détectés par le code résidu utilisé.

4.1. Le Diagramme Bloc

Le diagramme bloc des multiplieurs auto-contrôlables basés sur le code résidu est présenté dans la figure 4.1. Il est composé d'un bloc multiplieur, de deux générateurs modulo *A* pour générer les parties contrôle des opérandes d'entrée, d'un multiplieur modulo *A* pour générer la partie contrôle du résultat de la multiplication, un contrôleur arithmétique pour vérifier le résultat de la multiplication par rapport à la partie contrôle et finalement d'un générateur de parité pour générer la parité du résultat de la multiplication qui sera utilisée par les autres blocs du chemin de données. Le contrôleur arithmétique est constitué d'un générateur modulo *A* pour générer le résidu du résultat de multiplication, d'un bloc d'inverseurs, d'un translateur pour résoudre le problème de la double représentation du zéro et d'un contrôleur double-rail fournissant le signal d'erreur.
Pour ne pas pénaliser les performances du multiplieur, des latches (points mémoire) sont insérés à la sortie du multiplieur et à un certain niveau de la partie de contrôle. L'emplacement des latches divise le chemin critique de la circuiterie de contrôle en deux parties. Une avant les latches (temps de propagation T_2), et une après les latches (temps de propagation T_3). Dans la figure 4.1 on montre un emplacement qui correspond aux temps T_2 et T_3 représentés par les flèches en lignes continues. Les deux autres possibilités d'emplacement sont représentées par des flèches en lignes pointillées. Le résultat du multiplieur est contrôlé pendant le cycle d'horloge suivant. Dans ce cas, les performances du multiplieur ne sont pas pénalisées si aucune des parties placées avant ou après les latches n'a un temps de propagation supérieur au

temps de propagation du multiplieur lui-même. En se référant à la figure 4.1, les performances du multiplieur ne sont donc pas pénalisées si :

$T = Max(T_2, T_3, T_4) < T_1$.

Le choix de la position des latches dans la circuiterie de contrôle permet donc de jouer sur les valeurs de T2 et T3 (T4 étant fixe) de façon à obtenir la pénalisation de vitesse la plus faible possible.

Pour générer le multiplieur auto-contrôlable, nous avons implémenté des générateurs de macro-blocs paramétrables qui génèrent les différents blocs de la figure 4.1.

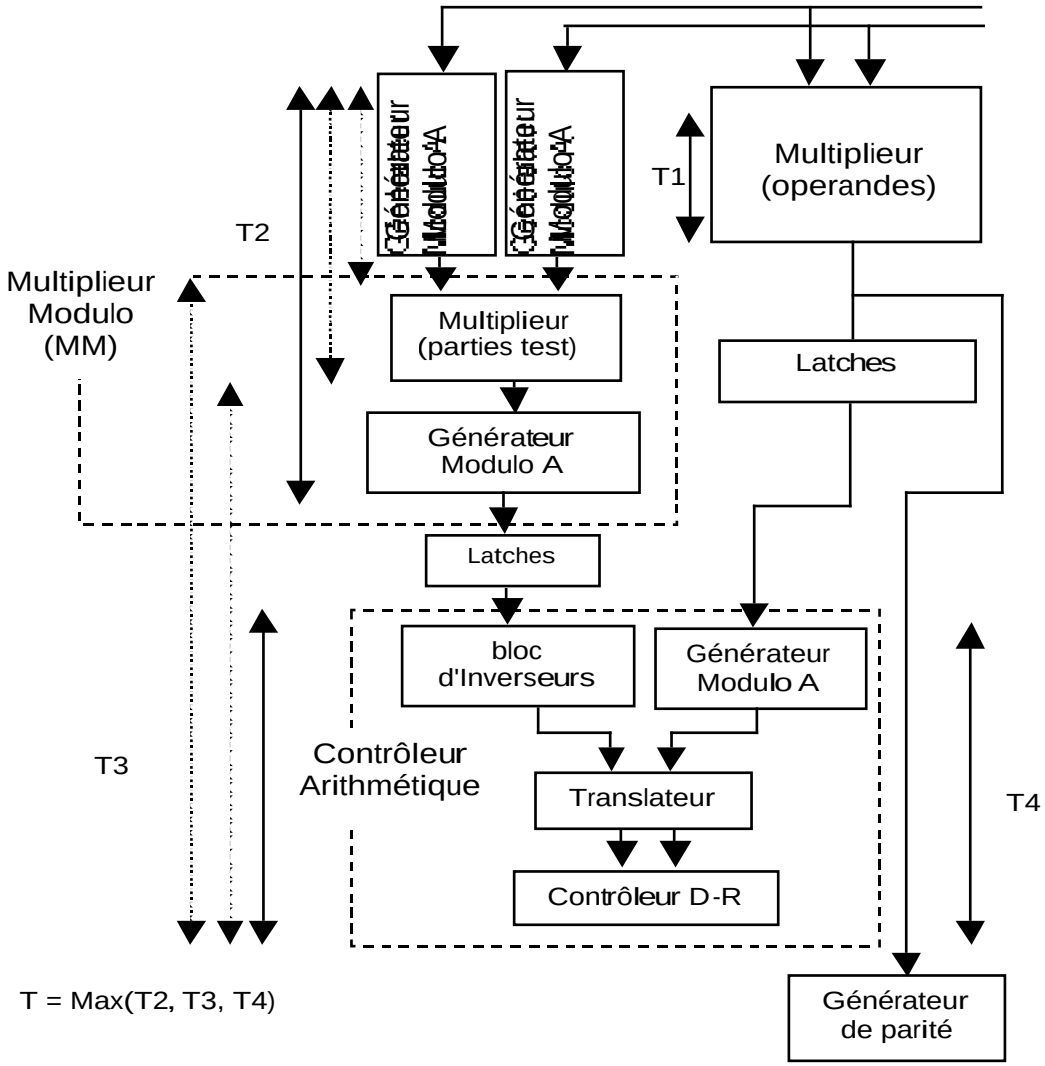


Figure 4.1. Le bloc diagramme d'un multiplieur auto-testable basé sur le code résidu

4.2. Multiplication Par Réseaux Cellulaires

Seuls les nombres non signés sont considérés ici. Considérons la multiplication de deux nombres *A* et *B* écrits en base 2 sur 8 bits. Les multiplieurs cellulaires sont constitués de cellules très simples, placées suivant un réseau d'interconnexion régulier, dont la topologie est calquée sur la disposition des termes A_iB_j (produits partiels) à la figure 4.2.

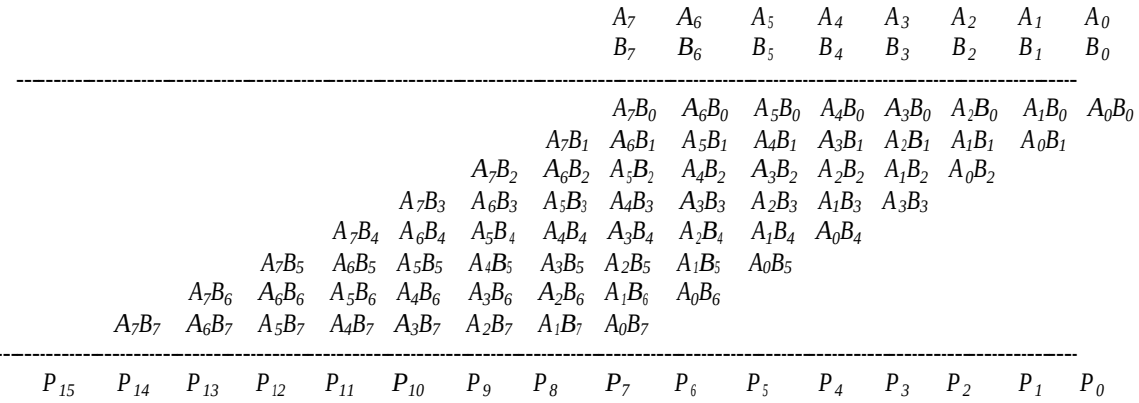


Figure 4.2 Une multiplication à la main

La cellule de base n'est rien d'autre qu'un FA. A l'aide de telles cellules, on peut construire un ou plusieurs additionneurs séquentiels pour calculer les sommes partielles de la figure 4.3. Il est toutefois préférable de calculer ces sommes partielles en notation "carry save", c'est à dire en conservant les retenues au lieu de les propager. En 1963, Braun [Bra63] a proposé un réseau très régulier, composé de FAs, et effectuant des multiplications en utilisant la notation "carry save" pour les résultats partiels. Un réseau permettant de multiplier des nombres écrits en base 2 sur n bits est constitué de n-1 additionneurs "carry save" (qui sont des rangs de cellules FA), et d'un additionneur séquentiel simple qui génère le résultat final. Ce réseau est présenté dans la figure 4.3 pour n = 8.

4.2.1. Analyse des Fautes pour les Multiplieurs Cellulaires

Pour un code de résidu, toutes les erreurs générées par une faute sont détectées si et seulement si la valeur arithmétique de chaque erreur générée par cette faute aux sorties primaires du multiplieur ne se divise pas par la base du code résidu utilisé. Pour un multiplieur cellulaire, cette propriété est facile à démontrer. Comme nous avons vu dans la section 3.2.1 une erreur produite sur les sorties d'une cellule de FA/HA a la valeur arithmétique $\pm 2^i$. Cette erreur est additionnée au résultat final via le réseau de cellules FA/HA du multiplieur pour donner ainsi la même valeur arithmétique d'erreur à la sortie du multiplieur. La valeur $\pm 2^i$ est un nombre

pair et elle ne se divise pas par 3. Par conséquent on peut utiliser la valeur 3 comme base du code résidu, qui est la solution la plus économique.

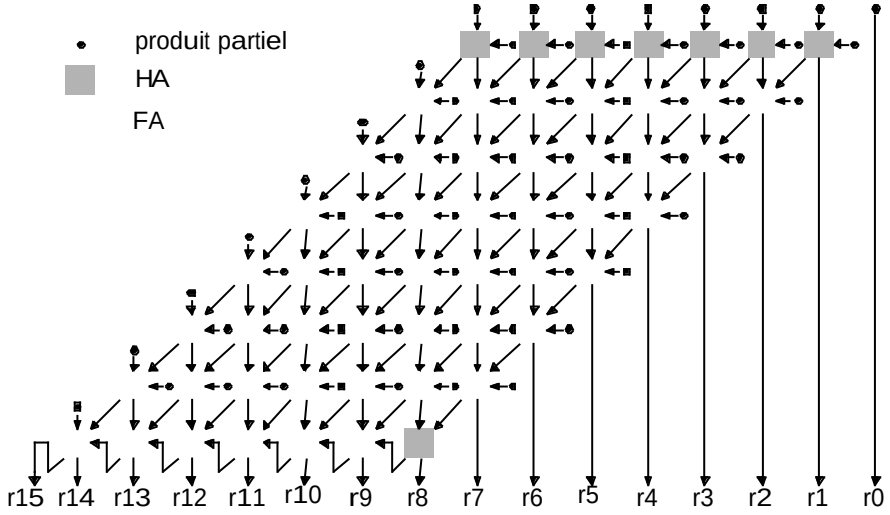


Figure 4.3. Multiplieur de Braun 8x8 bits

4.2.2. Implémentation et Résultats

Notre outil informatique a été utilisé pour obtenir les résultats des tableaux 4.1 et 4.2. Ces résultats sont basés sur la technologie CMOS ES2-1 μm.

Délai (ns)	Braun 8x8			Braun 16x16			Braun 32x32			Braun 64x64		
	T1(ns)	T(ns)	%	T1	T	%	T1	T	%	T1	T	%
FA type1	29,9	27,1	0,0	64,7	35,5	0,0	135	43,8	0,0	282,9	52,4	0,0
FA type2	34,4	27,1	0,0	74,1	35,5	0,0	155,3	43,8	0,0	321,6	52,4	0,0

Tableau 4.1 : Pénalisation des performances pour le multiplieur de Braun.

Coût (mm2)	Braun 8x8			Braun 16x16			Braun 32x32			Braun 64x64		
	Conv.	F.S.	%	Conv.	F.S.	%	Conv.	F.S.	%	Conv.	F.S.	%
FA type1	0,298	0,439	47,3	1,312	1,594	21,5	5,488	6,053	10,3	22,430	23,560	5,0
FA type2	0,259	0,400	54,4	1,128	1,410	25,0	4,701	5,266	12,0	19,179	20,309	5,9

Tableau 4.2 : Le surcoût pour le multiplieur de Braun.

Dans le tableau 4.1, T1 représente le temps de propagation maximum du multiplieur de Braun et T le temps de propagation maximum de la partie d'auto-contrôle comme indiqué dans la figure 4.1. On peut remarquer que T1 est toujours supérieur à T et par conséquent les performances du multiplieur de Braun ne sont pas pénalisées (colonne % égale toujours 0). Dans le tableau 4.2, la colonne Conv. donne la surface du multiplieur conventionnel, et la colonne FS donne la surface de la version sûre en présence de fautes. La colonne % donne le surcoût de la version FS par rapport à la version conventionnelle. La ligne FA type 1 concerne

les multiplieurs utilisant la cellule de FA de la figure 3.2, et la ligne FA type 2 concerne les multiplieurs utilisant la cellule de FA de la figure 3.3.

Le surcoût pour les multiplieurs 8x8 est élevé. Dans ce cas le schéma de prédiction de parité [Nic97] est meilleur (voir chapitre 5). Pour les autres cas, on obtient des résultats moins coûteux que pour le code de parité et on obtient même des coûts très faibles pour les grands multiplieurs. Ces coûts incluent tout le matériel nécessaire pour insérer le multiplieur dans un chemin de données contrôlé par la parité, à savoir un arbre pour générer le bit de parité du résultat.

4.3. Multiplication utilisant les Arbres de Wallace

Le délai du multiplieur de Braun est proportionnel aux tailles des opérandes d'une manière linéaire. L'utilisation des arbres de Wallace [Hen95] [Wal64] pour additionner les produits partiels permet d'obtenir un délai proportionnel aux tailles des opérandes d'une manière logarithmique.

La multiplication est réalisée en trois étapes. La première est la génération des produits partiels avec une rangée de portes logiques *AND*, et qui a un temps de propagation fixe. La seconde étape est un arbre de Wallace qui réduit les produits partiels en deux nombres binaires en un temps proportionnel au logarithme des tailles des opérandes d'entrée. La dernière étape est un additionneur qui additionne les deux nombres binaires produits par l'arbre de Wallace. Cet additionneur peut être séquentiel, mais on perd en vitesse de calcul. Un additionneur rapide qui a un temps de propagation logarithmique sera donc préférable.

4.3.1. Implémentation des Multiplieurs Basés sur les Arbres de Wallace

La structure des arbres de Wallace générés est expliquée par la suite à l'aide de la figure 4.6. La stratégie adoptée pour générer ces arbres est basée sur la méthode de Dadda [Dad65] [Mul89]. Elle consiste à regrouper les produits partiels de même poids en utilisant le plus petit nombre possible de cellules de *FA/HA*. Ce regroupement va produire les rangs (colonnes) de différents poids de l'arbre. Une cellule de *FA* recevant trois entrées de poids *i* génère deux sorties de poids *i* et *i+1* respectivement. La sortie de poids *i* sera utilisée dans le même rang de cellules de *FA/HA* que la cellule qui l'a générée (de poids i). La sortie de poids *i+1* sera utilisée dans le rang suivant dans l'arbre (de poids *i+1*). La dernière rangée des cellules de l'arbre produisent des paires de signaux de même poids (la retenue d'une cellule et la somme d'une cellule de rang supérieur). Les deux nombres binaires ainsi formés seront additionnés par un additionneur binaire.

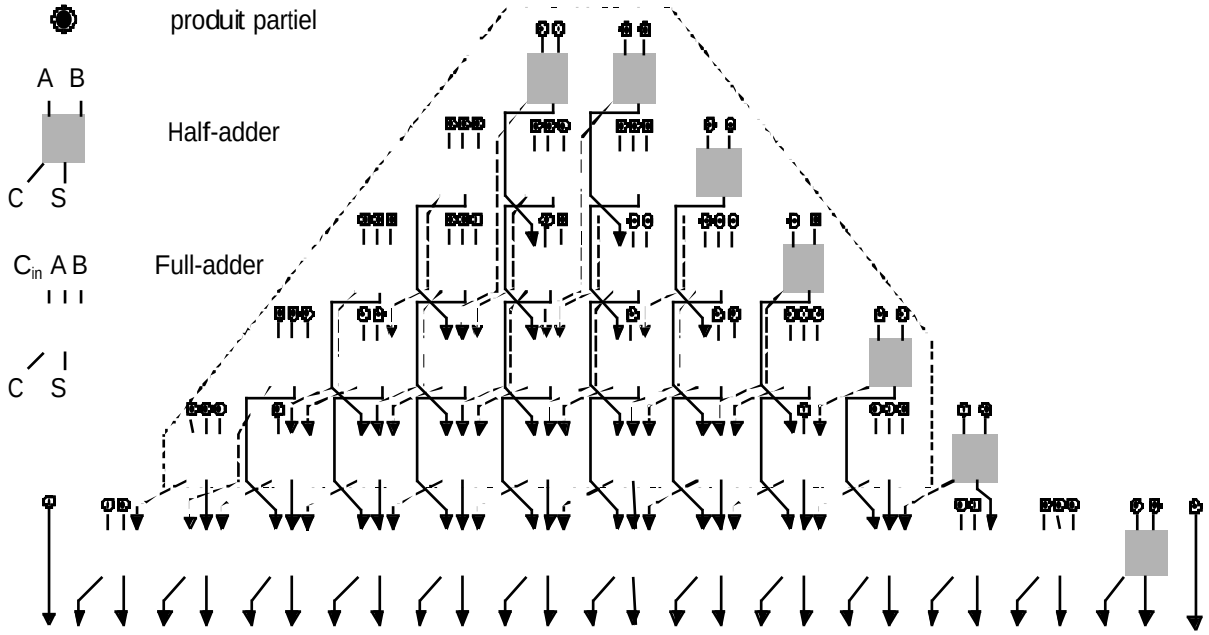


Figure 4.6. Arbre de Wallace 8x8

4.3.2. Analyse des Fautes pour les Multiplieurs utilisant les Arbres de Wallace

Bien que la structure du réseau de cellules de *FA/HA* des arbres de Wallace soit différente de celle des multiplieurs cellulaires, les erreurs produites à la sortie d'un arbre de Wallace et qui sont dues à une faute dans une cellule de *FA/HA* sont les mêmes. Dans les deux cas, le réseau de cellules *FA/HA* et l'additionneur final vont additionner la valeur arithmétique de l'erreur au résultat final de la multiplication. Par conséquent, la base *3* peut assurer la sûreté en présence de fautes pour toute faute affectant le réseau de cellules *FA/HA*. Mais, le choix de la base de détection d'erreurs dépend aussi de l'additionneur utilisé en aval de l'arbre de Wallace, car les fautes dans cette partie pourront générer des erreurs de valeurs différentes.

Cas 1. Additionneur séquentiel :

Dans ce cas, la valeur arithmétique finale pour une erreur locale dans l'additionneur est aussi de la forme $\pm 2^i$ (parce que l'additionneur est composé de cellules *FA/HA*). La base *3* garantit donc la sûreté en présence de fautes pour le multiplieur.

Cas 2. Additionneurs rapides :

L'utilisation d'un additionneur séquentiel (délai linéaire) va pénaliser la vitesse du multiplieur (bien que son coût soit minimum). Pour améliorer la vitesse, l'additionneur doit être un additionneur à calcul anticipé des retenues. Dans ce cas, la base nécessaire pour avoir la sûreté en présence de fautes du multiplieur sera celle imposée par l'additionneur. Le calcul de la base sera fait de la manière présentée dans les sections 3.3 et 3.4.

4.3.3 Implémentation et Résultats

Les résultats pour le multiplieur de Wallace avec plusieurs types d'additionneurs sont reportés dans les tableaux 4.3 et 4.4. Les bases de détections d'erreurs sont celles trouvées au chapitre 2. La technologie CMOS ES2-1 μm est utilisée.

Délai (ns)			Wallace 8x8			Wallace 16x16			Wallace 32x32			Wallace 64x64		
			T1	T	%	T1	T	%	T1	T	%	T1	T	%
Kogge-Stone	37	FA1				34,7	60,8	75,2	54,1	67,0	23,8	91,0	70,0	0,0
		FA2				36,3	60,8	67,5	55,5	67,0	20,7	92,2	70,0	0,0
	2^{12}-1	FA1	23,5	37,3 (2^6-1)	58,7	34,7	58,1	67,4	54,1	58,1	7,4	91,0	58,1	0,0
		FA2	25,8	37,3 (2^6-1)	44,6	36,3	58,1	60,1	55,5	58,1	4,7	92,2	58,1	0,0
Han-Carlson	2^6+1	FA1				37,8	47,7	26,2	57,3	56,9	0,0	94,1	60,2	0,0
		FA2				39,3	47,7	21,4	58,7	56,9	0,0	95,3	60,2	0,0
	2^8-1	FA1	24,3	28,3 (2^4-1)	16,5	37,8	48,1	27,2	57,3	56,4	0,0	94,1	65,3	0,0
		FA2	26,2	28,3 (2^4-1)	8,0	39,3	48,1	22,4	58,7	56,4	0,0	95,3	65,3	0,0
Brent-Kung	7	FA1	26,6	25,6	0,0	39,6	34,0	0,0	61,9	42,5	0,0	101,9	51,2	0,0
		FA2	27,3	25,6	0,0	40,3	34,0	0,0	62,4	42,5	0,0	102,3	51,2	0,0
Sklansky	3	FA1	24,1	27,1	12,4	36,4	35,5	0,0	56,9	43,8	0,0	95,3	52,4	0,0
		FA2	26,1	27,1	3,8	37,9	35,5	0,0	58,3	43,8	0,0	96,6	52,4	0,0
CLA	3	FA1	24,7	27,1	9,7	38,6	35,5	0,0	58,3	43,8	0,0	98,4	52,4	0,0
		FA2	26,3	27,1	3,0	39,3	35,5	0,0	59,4	43,8	0,0	98,8	52,4	0,0

Tableau 4.3 : Performances du multiplieur de Wallace.

Tout d'abord on peut remarquer la grande amélioration des performances des multiplieurs de wallace par rapport au multiplieur de Braun, le tout pour une faible augmentation de coût. Concernant les parties d'auto-contrôle les bases 3 et 7 utilisées dans trois cas des multiplieurs rapides n'affecte pas les performances du multiplieur (sauf pour 8x8), tandis que l'utilisation de bases plus grandes pénalise la performance. Cette pénalisation diminue avec l'augmentation des dimensions du multiplieur. D'une manière générale on peut dire qu'une version non auto-contrôlable plus rapide que d'autres versions non auto-contrôlables peut devenir moins rapide que ces dernières en passant à la version auto-contrôlable.

Dans le cas de l'additionneur de Kogge-Stone, bien que la base 2^{12}-1 est beaucoup plus grande que la base 37, elle donne nettement de meilleurs résultats. Si pour des raisons de vitesse l'additionneur de Kogge-Stone est utilisé, la base 2^{12}-1 sera donc préférable. Dans le cas de l'additionneur de Han-Carlson, les bases 2^8-1 et 2^6+1 donnent à peu près les mêmes résultats.

Surcoût (mm2)			Wallace 8x8			Wallace 16x16			Wallace 32x32			Wallace 64x64		
			Conv.	F.S	%	Conv.	F.S	%	Conv.	F.S	%	Conv.	F.S	%
Kogge-Stone	37	FA1				1,505	2,855	89,7	5,943	7,809	31,4	23,583	26,012	10,3
		FA2				1,346	2,696	100,3	5,206	7,072	35,8	20,434	22,863	11,9
	2^{12}-1	FA1	0,364	0,641 (2^6-1)	76,1	1,505	2,431	61,5	5,943	7,157	20,4	23,583	25,432	7,8
		FA2	0,336	0,613 (2^6-1)	82,4	1,346	2,272	68,8	5,206	6,420	23,3	20,434	22,283	9,0
Han-Carlson	2^6+1	FA1				1,471	2,030	38,0	5,878	6,777	15,3	23,352	24,893	6,6
		FA2				1,311	1,870	42,6	5,142	6,041	17,5	20,203	21,744	7,6
	2^8-1	FA1	0,358	0,539 (2^4-1)	50,6	1,471	2,005	36,3	5,878	6,725	14,4	23,352	24,843	6,4
		FA2	0,329	0,510 (2^4-1)	55,0	1,311	1,845	40,7	5,142	5,989	16,5	20,203	21,694	7,4
Brent-Kung	7	FA1	0,340	0,506	48,8	1,406	1,730	23,0	5,689	6,322	11,1	22,851	24,086	5,4
		FA2	0,311	0,477	53,4	1,246	1,570	26,0	4,952	5,585	12,8	19,702	20,937	6,3
Sklansky	3	FA1	0,350	0,491	40,3	1,440	1,722	19,6	5,820	6,385	9,7	23,178	24,308	4,9
		FA2	0,321	0,462	43,9	1,280	1,562	22,0	5,083	5,648	11,1	20,029	21,159	5,6
CLA	3	FA1	0,338	0,479	41,7	1,402	1,684	20,1	5,679	6,244	9,9	22,828	23,958	5,0
		FA2	0,309	0,450	45,6	1,242	1,524	22,7	4,943	5,508	11,4	19,679	20,809	5,7

Tableau 4.4 : Surcoût pour le multiplieur de Wallace.

Contrairement aux multiplieurs de Braun, l'utilisation du FA de type1 au lieu du FA de type 2 n'améliore que très peu la vitesse des multiplieurs. Le FA de type 2, de coût plus faible, sera toujours utilisé dans le cas des arbres de Wallace implémentés comme on vu précédemment.
Finalement, la vitesse et le coût détermineront le choix de l'additionneur à utiliser. Le coût des multiplieurs de Wallace avec et sans auto-contrôle est présenté dans le tableau 4.4. Les cas de multiplieurs utilisant les codes à base 3 donnent les plus faibles augmentations de surface (CLA et Sklansky) suivis par le cas de Brent-Kung (base 7). Ces trois cas donnent des coûts plus faibles que la parité à partir des tailles de multiplieurs de 16x16. Le cas de Han-Carlson donne des coûts plus faibles que la parité à partir des tailles de multiplieurs de plus de 16x16. Dans tous les cas on a des coûts très faibles pour des multiplieurs de tailles 64x64. Les tableaux 4.3 et 4.4 permettent de sélectionner la structure de multiplieur la plus intéressante selon les coûts en surface et les performances. Ainsi, pour un multiplieur 32x32, la solution Kogge-Stone est la plus rapide pour un multiplieur conventionnel. Mais pour un multiplieur auto-contrôlable, la solution Sklansky est plus rapide (56,9 ns au lieu de 58,1 en utilisant la cellule FA1). Par ailleurs, la solution Sklansky auto-contrôlable est moins coûteuse que la solution Kogge-Stone auto-contrôlable (6,385 mm^2 au lieu de 7,157 mm^2). On va donc choisir

sans hésitation la solution Sklansky. Ceci n'est pas le cas pour les multiplieurs conventionnels. Dans ce cas la solution Sklansky est plus économique en surface que la solution Kogge-Stone (5,820 mm^2 au lieu de 5,943 mm^2) mais elle est moins rapide (56,9 ns contre 54,1 ns). Ainsi, dans le cas des multiplieurs conventionnels on pourra choisir la solution Kogge-Stone pour des raisons de vitesse.

4.4. Multiplieurs Basés sur le Codage de Booth

En 1951, A. D. Booth a présenté une technique de codage pour la multiplication de nombres binaires signés. Cette technique est aujourd'hui utilisée dans un grand nombre de multiplieurs [Mor91] [Sat91] [Bur94] [Yu95] [Mak96]. L'algorithme de Booth réduit le nombre de produits partiels en codant le multiplieur (*A*). A l'origine, l'algorithme de Booth réalise le codage en série [Boo51]. Un algorithme de Booth modifié (Booth2) permet de réaliser le codage en parallèle [McS61] ; C'est l'algorithme utilisé aujourd'hui. Booth2 réduit le nombre de produits partiels de n^2 en $n\ (n+1)\ /\ 2$. Mais cette réduction est en partie diminuée par la complexité du circuit de sélection des produits partiels. Dans l'algorithme Booth2, le multiplieur (*A*) est divisé en groupes chevauchés de trois bits. Chacun de ces groupes est codé en parallèle pour sélectionner un seul produit partiel selon la table 4.5. Dans la table 4.6 est présentée la relation entre les produits partiels et les bits codés, et la table 4.7 représente la table de vérité des produits partiels.

Bits du Multiplieur (A)	Sélection
000	+0
001	+B
010	+B
011	+2B
100	-2B
101	-B
110	-B
111	-0

Table 4.5. Sélection du produit partiel

Les digits codés du Multiplieur	Produit Partiel									+	Remarques
	pp_8	pp_7	pp_6	pp_5	pp_4	pp_3	pp_2	pp_1	pp_0		
0 (‘0’)	0	0	0	0	0	0	0	0	0	0	-
+1 (‘p1’)	b_7	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	0	B
-1 (‘m1’)	$\overline{b_7}$	$\overline{b_7}$	$\overline{b_6}$	$\overline{b_5}$	$\overline{b_4}$	$\overline{b_3}$	$\overline{b_2}$	$\overline{b_1}$	$\overline{b_0}$	1	Complémenter B & ajouter 2^0
+2 (‘p2’)	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	b_{-1}	0	Décaler B d'une position à gauche
-2 (‘m2’)	$\overline{b_7}$	$\overline{b_6}$	$\overline{b_5}$	$\overline{b_4}$	$\overline{b_3}$	$\overline{b_2}$	$\overline{b_1}$	$\overline{b_0}$	$\overline{b_{-1}}$	1	Décaler B & ajouter 2^0

Table 4.6. Relation entre les produits partiel et les bits codés

Les digits codés du Multiplieur	Sortie		ajouter 2^0
0 (‘0’)	$pp_i = 0$	pour i=1, ..., 8	0
+1 (‘p1’)	$pp_i = b_i$	pour i=1, ..., 8	0
-1 (‘m1’)	$pp_i = b_i$	pour i=1, ..., 8	1
+2 (‘p2’)	$pp_i = b_{i-1}$	pour i=1, ..., 8	0
-2 (‘m2’)	$pp_i = b_{i-1}$	pour i=1, ..., 8	1

Table 4.7. Table de vérité des produits partiels

Il est important de bien noter que $b_8 = b_7$ est le bit de signe de (*A*), et que pp_8 est le bit de signe d'une ligne de produits partiels.

La figure 4.7 représente un exemple de multiplication de nombres signés utilisant l'algorithme Booth2. Les opérandes *A* et *B*, et le résultat sont écrits en notation de complément à *2*. La figure 4.8 représente un exemple de multiplication non signée utilisant le même algorithme.

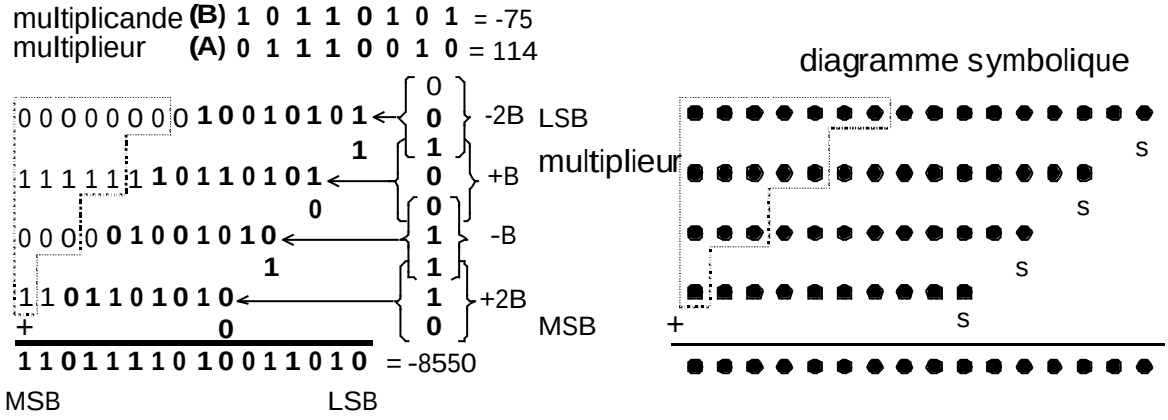


Figure 4.7. Une multiplication signée 8x8 de Booth2

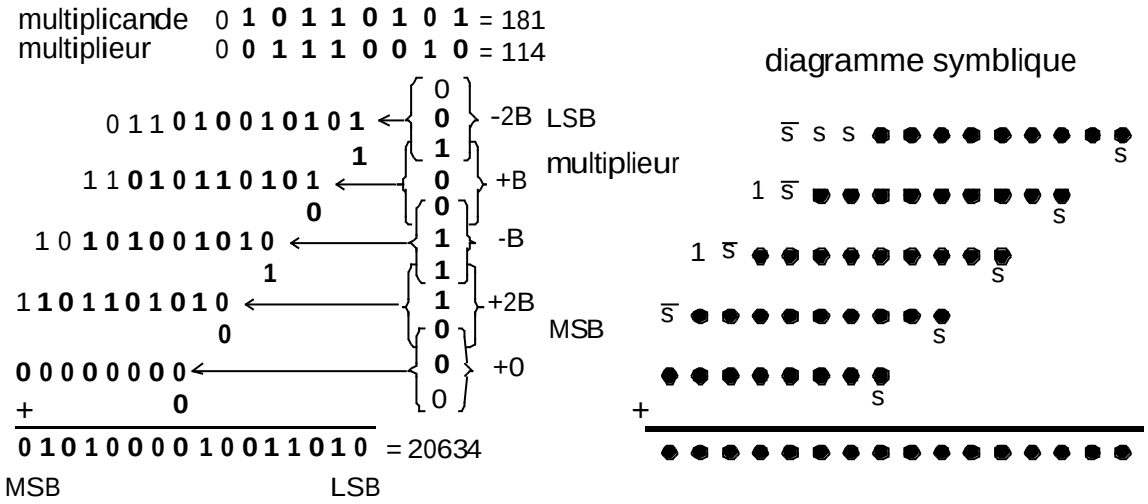


Figure 4.8. Une multiplication non signée 8x8 de Booth2

On remarque que pour réaliser une multiplication non signée, un bit supplémentaire est nécessaire pour chacune des deux opérandes. Dans la littérature, il y a aussi les algorithmes Booth3 et Booth4 qui sont proposés, mais ils n'apportent pas d'améliorations significatives par rapport à Booth2 à cause de la complexité des circuits de codage et de sélection et du manque de régularité du routage pour des technologies différentes [Twa95]. De fait, on n'a considéré que l'algorithme de Booth2. On va à présent étudier de plus près les différents schémas des circuits de codage et de sélection. La sélection du bit correct de la multiplicande dépend des quatre signaux m_1, m_2, p_1 et p_2 qui veulent dire moins 1, moins 2, plus 1, et plus 2 respectivement. Les équations (1) et (2) sont ainsi évidentes.

$$pp_j = (p_1 b_j) + (p_2 b_{j-1}) + (m_1 \overline{b_j}) + (m_2 \overline{b_{j-1}}) \quad (1)$$

$$add_1 = m_1 + m_2 \quad (2)$$

La figure 4.9 montre l'implémentation correspondante aux équations (1) et (2) [Ara89], où les signaux m_1, m_2, p_1 et p_2 sont générés par une cellule de codage.

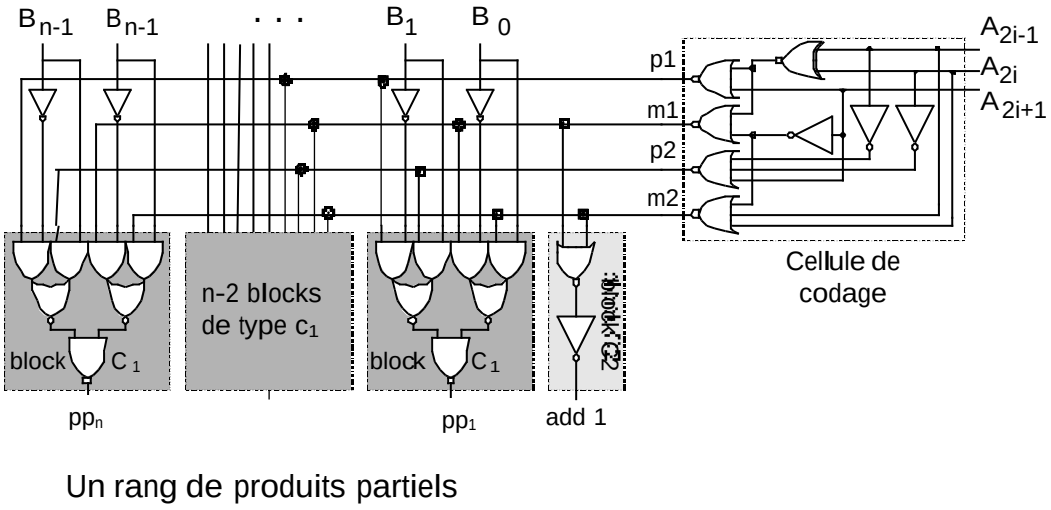


Figure 4.9. Une première implémentation du codage et de la sélection pour le Booth2

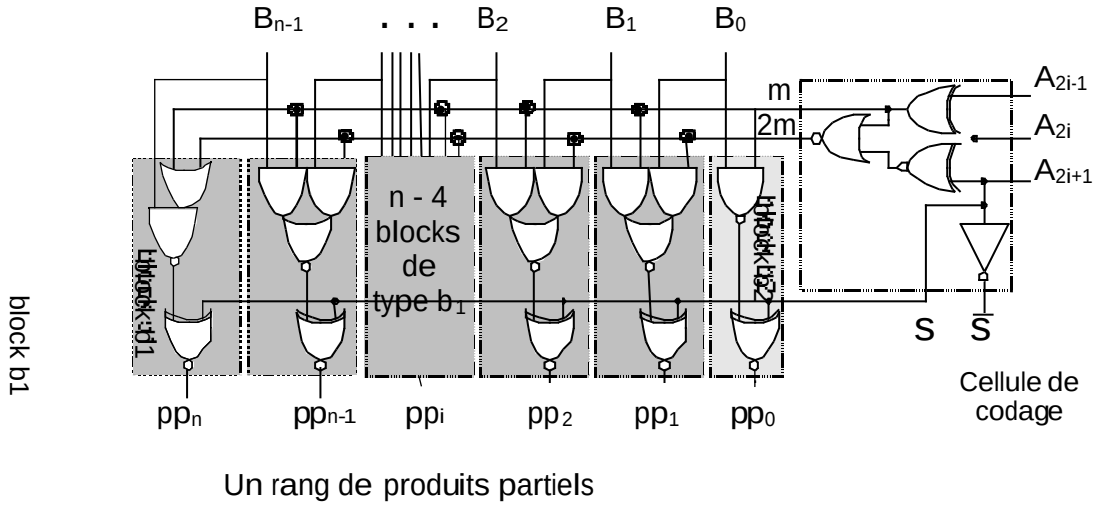


Figure 4.10. Une version compacte des circuits de codage et de sélection

La figure 4.10 représente une version plus compacte des circuits de codage et de sélection [Wes94]. Ce schéma introduit un léger retard par rapport au schéma précédent. Les signaux m, 2m et s représentent respectivement : ajouter la multiplicande, multiplier la multiplicande par deux et complémenter les bits obtenus.

4.4.1. Le Codage de Booth avec les Topologies en Notation "Carry Save"

En notation "carry save", la topologie des cellules de *FA/HA* est calquée sur la disposition des produits partiels à la figure 4.8. Sur le schéma résultant on peut remarquer que l'extension des signes des produits partiels est coûteuse en matière de surface. Ce problème peut être résolu en utilisant une de deux techniques différentes qui sont la propagation de signe et la génération de signe respectivement [Ara89]. Dans cette section, on va détailler la technique de propagation de signe. L'étude de la sûreté en présence de fautes est la même pour les deux techniques. La technique de génération de signe sera développée dans la section 4.4.2. La technique de propagation de signe consiste à propager le signe (le bit de poids le plus significatif) de chaque ligne (somme partielle) vers les deux bits de poids les plus significatifs de la ligne suivante. La figure 4.11.a représente un multiplieur de Booth2 8x8 utilisant les

circuits de codage et de sélection de la figure 4.9 et la technique de propagation de signe. La figure 4.11.b représente le même multiplieur pour les circuits de codage et de sélection de la figure 10. Les carrés en gris représentent des cellules de *HA*, et ceux en blanc représentent des cellules de *FA*. Le bloc c_1 est la cellule de sélection et le bloc c_2 implémente l'équation (2). L'additionneur en dernière étape est un additionneur séquentiel (cet additionneur peut être remplacé par un additionneur rapide).

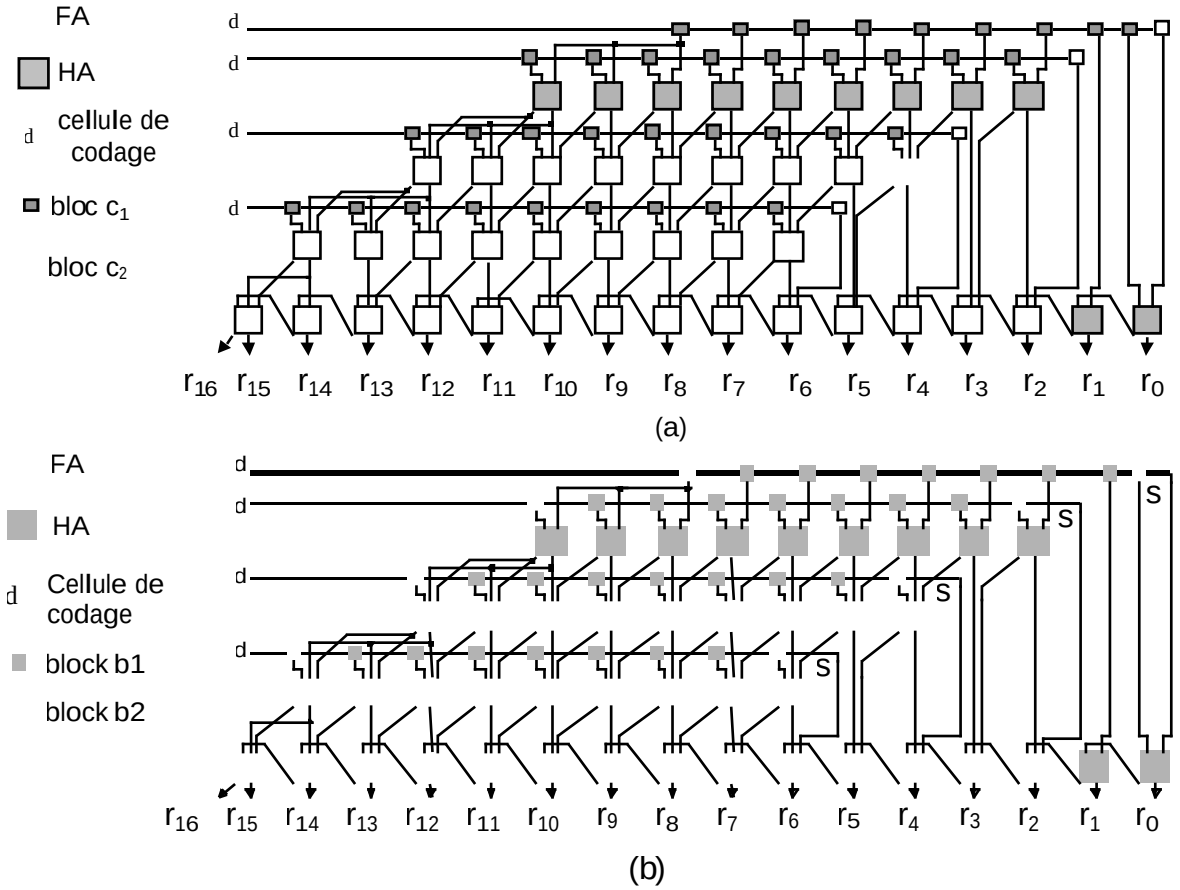


Figure 11 Implémentations en notation "carry save" d'un multiplieur de Booth2 8x8

4.4.1.1. Analyse en Présence de Fautes

L'analyse est faite par rapport aux schémas de la figure 4.11. Les cellules de *FA/HA* considérées sont celles vues au chapitre 3. Les opérandes d'entrée *A* et *B* sont de *n* bits chacune, et le résultat de multiplication *R* est de *2n* bits. Pour faire l'analyse on distingue trois type de fautes : la faute dans une cellule de codage, la faute dans une cellule de sélection et la faute dans une cellule de *FA/HA*. Une faute cellulaire introduit une erreur locale qui se propage vers la sortie du multiplieur pour donner une erreur globale. Pour déterminer la valeur de cette erreur globale il faut étudier s'il y a débordement ou pas.

Faute dans une cellule de codage

Dans la figure 5, une faute dans une cellule de codage affecte une ligne entière de produits partiels sélectionnés. La valeur arithmétique d'une telle erreur locale est aléatoire. Elle dépend de la faute dans la cellule de codage et de la valeur de l'opérande *B*. La détection de telles fautes avec le code résidu est très coûteuse (il faut une base de valeur > 2^n). La solution

adoptée pour remédier à ce problème est la duplication du circuit de codage. Un contrôleur double-rail est ensuite utilisé pour fournir un signal d'erreur locale.

Faute dans une cellule de sélection

Une faute dans une cellule de sélection introduit une erreur locale de valeur arithmétique simple ($\pm 2^i$). Cette erreur locale va être additionnée au résultat final R pour donner une erreur globale. S'il n'y a pas de débordement, l'erreur globale aura la même valeur que l'erreur locale et donc détectable avec la base 3 dans ce cas.

Faute dans une cellule de FA/HA [Spa90]

On distingue deux groupes différents de cellules de *FA/HA*. Le premier groupe est constitué de l'ensemble des cellules qui se trouvent à l'extrémité gauche de chaque ligne. Le deuxième groupe est constitué du reste des cellules.

Une faute dans une cellule du premier groupe qui se trouve à la colonne de poids i introduit une erreur locale de valeur arithmétique $\pm \propto\_2^i$ ($\propto = 1, 2$). L'erreur globale résultante est la même comme si toutes les cellules de la même ligne et de poids $k \geq i$ dans le réseau d'origine (sans propagation de signe) étaient affectées de la même faute. La valeur de l'erreur e qui sera additionnée au résultat final R est donc :

$e = [\pm \propto\_(2^i + 2^{i+1} + \_\_\_ + 2^{2n-1})] \bmod 2^{2n}$

$e = [\pm \propto\_(2^{2n} - 2^i)] \bmod 2^{2n}$

$e = \mathtt{m} \propto\_2^i$.

Une faute dans une cellule du deuxième groupe introduit donc une erreur locale de la forme $\pm \propto\_2^i$. Cette erreur se propage vers la sortie en gardant la même valeur. Il faut noter que ce type de fautes n'existe pas dans les multiplieurs de Booth utilisant la technique de génération de signe.

Considérons maintenant le cas où une erreur locale génère un débordement. Soit e la valeur de l'erreur à additionner à la sortie du multiplieur. L'analyse précédente a montré que l'erreur est toujours de la forme $\pm \propto\_2^i$ ($\propto = 1, 2$). Un débordement se produit lorsque :

$R + e \notin [-2^{2n-1} : 2^{2n-1}-1]$

où $[-2^{2n-1} : 2^{2n-1}-1]$ est l'intervalle de nombres représentables sur $2n$ bits en complément à deux, et R est le résultat de multiplication sans erreur. D'autre part on sait que :

les opérandes $A, B \in [-2^{n-1} : 2^{n-1}-1]$ et par conséquent $A\_B \in [-2^{2n-2} + 2^{n-1} : 2^{2n-2}]$.

De $R \leq 2^{2n-2}$, on déduit que la seule possibilité d'avoir un débordement positif a lieu pour :

$R + e > 2^{2n-1} - 1 \Rightarrow e \geq 2^{2n-1} - 1 - R$. R peut s'écrire sous la forme $R = 2^{2n-2} - x$ avec $x \geq 0$, ce qui donne $e > 2^{2n-1} - 1 - (2^{2n-2} - x)$

sachant que e est de la forme 2^i on trouve :

$2^i > 2^{2n-2} - 1 + x$

Par conséquent la seule possibilité d'avoir un débordement positif est pour $x = 0$ (si x > 0 le bit de poids 2^{2n-2} est à zéro et un débordement n'est plus possible) et $e = 2^{2n-2}, 2^{2n-1}$. On trouve alors :

$|(R + e) - R| = |(2^{2n-2} + 2^{2n-2}) - (2^{2n-2})|$

$= |(-2^{2n-1}) - (2^{2n-2})|$

$= 3\_2^{2n-2}$ pour la première valeur, et :

$|(R + e) - R| = |(2^{2n-2} + 2^{2n-1}) - (2^{2n-2})|$

$= |(2^{2n-2} - 2^{2n-1}) - (2^{2n-2})|$

$= 2^{2n-1}$ pour la deuxième valeur.

De même, un débordement négatif a lieu lorsque :

$R + e < -2^{2n-1}$. En écrivant R sous la forme $R = -2^{2n-2} + 2^{n-1} + x$ avec $x \geq 0$, on trouve :

$e < -2^{2n-2} - 2^{n-1} - x$

et donc $e = -2^{2n-1}$. Par conséquent :
$|(R + e) - R| = |(-2^{2n-1} - 2^{2n-2} + 2^{n-1} + x) - (-2^{2n-2} + 2^{n-1} + x)|$
$|(R + e) - R| = |(-2^{2n} + 2^{2n-2} + 2^{n-1} + x) - (-2^{2n-2} + 2^{n-1} + x)|$
$|(R + e) - R| = 2^{2n} - 2^{2n-1} = 2^{2n-1}$.

On peut résumer l'étude précédente :
le seul cas où une erreur locale de valeur arithmétique simple génère une erreur globale de valeur arithmétique non simple a lieu lorsque $A = B = -2^{n-1}$ et $e = 2^{2n-2}$.
En négligeant la possibilité d'avoir ce cas, on peut utiliser la base 3 pour garantir la sûreté en présence de fautes. Dans le cas contraire, la base 7 sera utilisée.
Si l'additionneur séquentiel est remplacé par un additionneur plus complexe, il faut tenir compte de la base appropriée pour détecter les fautes aussi présentes dans l'additionneur.

4.4.2. Utilisation des Arbres de Wallace

Dans le but d'améliorer la vitesse, les produits partiels codés peuvent évidemment être additionnés en utilisant un arbre de Wallace. On a vu précédemment qu'il y a deux méthodes pour réduire la surface en cas d'utilisation du codage de Booth : la méthode de propagation de signe que nous avons détaillée, et la méthode de génération de signe. La génération d'un arbre de Wallace utilisant la deuxième méthode est plus facile. Pour cette raison, dans cette section on va développer la méthode de génération de signe. La méthode de génération de signe est représentée par l'exemple de la figure 4.12.

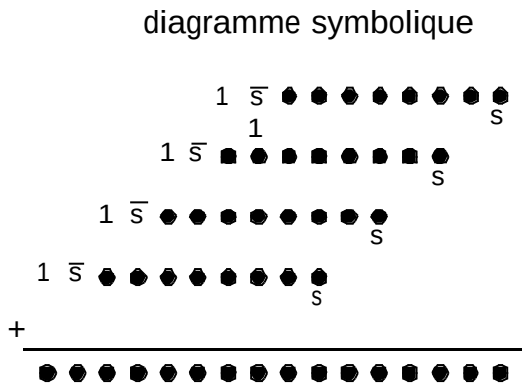


Figure 4.12. Exemple de la méthode de génération de signe sur 8x8

Cette méthode consiste à calculer le bit de signe. On démontre que le bit de signe peut être obtenu en effectuant les opérations suivantes :
- Complémenter le bit de signe de chaque produit partiel.
- Additionner 1 à gauche du bit de signe de chaque produit partiel.
- Additionner 1 au 9^{eme} bit du premier produit partiel.

La figure 4.13 représente un exemple de multiplieur de Booth 8x8, basé sur la génération de signe et utilisant un arbre de Wallace pour additionner les produits partiels.

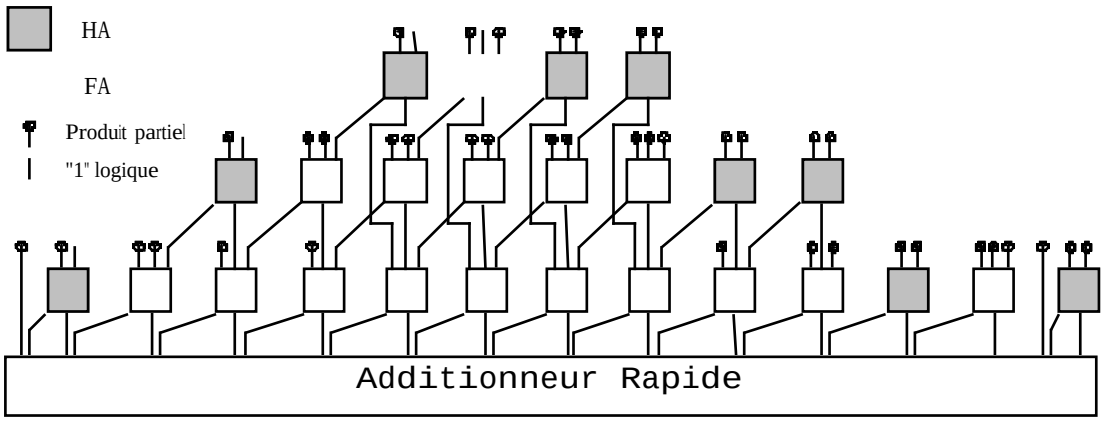


Figure 4.13. Un multiplieur de Booth 8x8 utilisant un arbre de Wallace

Les simplifications sur les cellules de FA/HA ayant un 1 logique à l'entrée ne sont pas reportées sur la figure, mais il faut remarquer qu'une erreur sur les cellules simplifiées génère des fautes simples.

4.4.3. Implémentation et Résultats

Dans cette section on reporte seulement les résultats pour le codage de Booth2 avec les arbres de Wallace (qui sont plus intéressants au niveau vitesse). La technologie CMOS ES2-1 µm est utilisée pour synthétiser.
On a vu dans l'analyse de fautes que la base minimale pour garantir la sûreté en présence de fautes est la base 7. Mais en négligeant la probabilité qu'une erreur locale simple génère une erreur de sortie non simple (on a vu que cette probabilité est très faible), la base 3 peut être utilisée. Toute fois, dans les cas des additionneurs de CLA et Sklansky, les résultats sont reportés pour les deux bases. Pour les autres cas d'additionneurs, la base est supérieure à 7. Elle couvre donc les erreurs arithmétiques non simples générées par la structure de Booth.

Type d'additionneur et la base utilisée		8x8			16x16			32x32			64x64		
		T1(ns)	T (ns)	%	T1	T	%	T1	T	%	T1	T	%
Kogge-Stone	37				34,0	60,8	78,8	46,2	67,0	45,0	67,3	70,0	4,0
	2^{12}-1	25,9	37,3 (2^6-1)	44,0	34,0	58,1	70,9	46,2	58,1	25,8	67,3	58,1	0,0
Han-Carlson	2^6+1				37,5	47,7	27,2	49,4	54,2	9,7	70,4	60,2	0,0
	2^8-1	26,8	28,3 (2^4-1)	5,6	37,5	48,1	28,3	49,4	56,4	14,2	70,4	65,3	0,0
Brent-Kung	7	27,7	25,6	0,0	39,8	34,0	0,0	54,3	42,5	0,0	78,4	51,2	0,0
Sklansky	3	26,0	27,1	4,2	36,0	35,5	0,0	49,0	43,8	0,0	71,6	52,4	0,0
	7	26,0	25,6	0,0	36,0	34,0	0,0	49,0	42,5	0,0	71,6	51,2	0,0
CLA	3	25,9	27,1	4,6	39,7	35,5	0,0	50,8	43,8	0,0	75,2	52,4	0,0
	7	25,9	25,6	0,0	39,7	34,0	0,0	50,8	42,5	0,0	75,2	51,2	0,0

Tableau 4.8 : Performances du multiplieur de Booth2-Wallace.

On peut remarquer que le coût des multiplieurs utilisant le codage de Booth est plus faible que le coût des multiplieurs de Braun et Wallace. Ceci est valable pour les versions conventionnelles mais aussi pour les versions auto-contrôlables. Pour les grandes dimensions (32x32 et 64x64), le coût des versions auto-contrôlables des multiplieurs de Booth est même plus faible que celui des versions simples des autres multiplieurs. D'autre part, la vitesse des multiplieurs de Booth-Wallace est nettement meilleure (20 à 30% plus rapides) que les multiplieurs de Wallace.
En raison de la duplication du circuit de codage et de l'utilisation d'un contrôleur double-rail pour détecter d'éventuelles erreurs, le surcoût en pourcentage de surface dans le cas des multiplieurs de Booth est supérieur au surcoût des autres cas.

Type d'additionneur et la base utilisée		8x8			16x16			32x32			64x64		
		simple	F.S	%	simple	F.S	%	simple	F.S	%	simple	F.S	%
Kogge-Stone	37				1,112	2,522	126,8	3,920	5,909	50,7	14,652	17,328	18,3
	2^{12}-1	0,331	0,637 (2^6-1)	92,4	1,112	2,099	88,8	3,920	5,257	34,1	14,652	16,748	14,3
Han-Carlson	2^6+1				1,088	1,707	56,9	3,851	4,872	26,5	14,470	16,217	12,1
	2^8-1	0,322	0,403 (2^4-1)	25,2	1,088	1,683	54,7	3,851	4,821	25,2	14,470	16,157	11,7
Brent-Kung	7	0,308	0,501	63,0	1,029	1,414	37,4	3,660	4,441		13,980	15,412	10,2
Sklansky	3	0,312	0,482	54,5	1,056	1,399	32,5	3,790	4,478	18,2	14,292	15,625	9,3
	7	0,312	0,507	62,5	1,056	1,441	36,5	3,790	4,546	19,9	14,292	15,729	10,1
CLA	3	0,306	0,476	55,6	1,025	1,368	33,5	3,661	4,349	18,8	13,956	15,286	9,5
	7	0,306	0,501	63,7	1,025	1,41	37,6	3,661	4,417	20,7	13,956	15,39	10,3

Tableau 4.9 : Surcoût pour les multiplieurs Booth2-Wallace

4.5. Réduction du Coût

On a vu que le choix de la base de détection d'erreur dépend de l'additionneur rapide utilisé et plus cette base est complexe plus le surcoût est élevé. Dans [Nic93] une solution pour tester les retenues d'un additionneur rapide basé sur la prédiction de parité (voir chapitre 5) a été proposée. Cette technique consiste à générer des retenues d'auto-contrôle à partir des retenues normales de l'additionneur en utilisant la relation suivante :

$C'_i = A_i \_ B_i + P_i \_ C_{i-1}$.

Les termes C_{i-1} sont les retenues normales de l'additionneur et les termes C'_i sont les retenues de test. A_i et B_i sont les bits à additionner de poids i et P_i le terme de propagation. Un contrôleur double-rail permet ensuite de comparer les retenues normales et les retenues d'auto-contrôle pour détecter d'éventuelles erreurs.

Dans le cas des multiplieurs utilisant un additionneur rapide, cette technique peut être composée avec le code résidu de base 3 pour éviter l'utilisation de bases de détection d'erreurs complexes. La relation de génération des retenues d'auto-contrôle peut dans ce cas être simplifiée pour donner :

$C'_i = G_i + P_i \_ C_{i-1}$.

Où les P_i et G_i sont respectivement les propagation et génération numéros i de l'additionneur. Bien que les P_i et G_i de l'additionneur sont les mêmes que ceux utilisés pour générer les retenues d'auto-contrôle, le schéma résultant est sûre en présence de fautes [Nic93]. Toutefois, les Gi doivent être dupliqués [Nic93]. Dans notre cas, une faute sur un G_i se propage vers les retenues d'auto-contrôle et les retenues normales en même temps et par conséquent elle ne sera pas détectée par le contrôleur double-rail. Néanmoins, une telle faute génère une erreur

arithmétique de valeur simple et sera donc détectée par le code résidu de base 3. En résumé, une faute (simple ou multiple) dans le bloc de génération des retenues de l'additionneur est détectée par le contrôleur double-rail et une faute simple dans le reste du multiplieur est détectée par le code résidu.
Les tableaux 4.10, 4.11 et 4.12 montrent une comparaison de coût entre la technique décrite ci-dessus et l'utilisation d'une base complexe ou de grande valeur dans le cas du multiplieur de Wallace (FA type 2) avec l'additionneur de Kogge-Stone et Han-Carlson.

Surcoût (%)	8x8	16x16	32x32	64x64
Base = 2^{12}-1	82.4 (2^6-1)	68.8	23.3	9.0
Base 3 + retenues d'auto-contrôle	52,7	27,3	14,8	7,3
Duplication du bloc de retenues	69,6	39,2	23,1	12,6

Tableau 4.10. Réduction du coût dans le cas du multiplieur de Wallace utilisant l'additionneur de Kogge-Stone

Surcoût (%)	8x8	16x16	32x32	64x64
Base = 2^8-1	55,0 (2^4-1)	40,7	16,5	7,4
Base 3 + retenues d'auto-contrôle	52,3	28,1	14,5	7,4

Tableau 4.11. Réduction du coût dans le cas du multiplieur de Wallace utilisant l'additionneur de Han-Carlson

Surcoût (%)	8x8	16x16	32x32	64x64
Base = 2^{12}-1	92,4 (2^6-1)	88,8	34,1	14,3
Base 3 + retenues d'auto-contrôle	63,7	39,8	23,0	12,3

Tableau 4.12. Réduction du coût dans le cas du multiplieur de Booth2 Wallace utilisant l'additionneur de Kogge-Stone

On peut remarquer une diminution importante du coût en surface surtout pour le multiplieurs de petites et moyennes tailles. La table 4.10 montre aussi que l'utilisation de la parité pour contrôler les retenues de l'additionneur est bien meilleure que la duplication du bloc de retenues.

4.5. Récapitulatif des Outils de Génération des Multiplieurs Auto-Contrôlables à code résidu

La figure 4.14 montre le flot de données de nos outils de génération de multiplieurs auto-contrôlables. L'utilisateur spécifie la taille du multiplieur ainsi que sa structure (cellulaire, arbre de Wallace, avec ou sans codage de Booth et le type de l'additionneur final). L'outil génère le multiplieur conventionnel suivant ces spécifications. Ensuite, le module d'analyse

d'erreurs génère toutes les valeurs arithmétiques d'erreurs possibles produites par cette structure pour les fautes de type collage. L'outil de génération de base détermine les bases des codes résidus détectant toutes ces erreurs. Les générateurs des différents macro-blocs génèrent les différents circuits d'auto-contrôle correspondant à la base choisie. Un outil de CAO permet à l'utilisateur d'estimer le coût en vitesse et en surface du multiplieur auto-contrôlable obtenu. Ces opérations peuvent être répétées pour une autre structure de multiplieur afin de sélectionner la structure offrant le meilleur compromis coût matériel/vitesse.

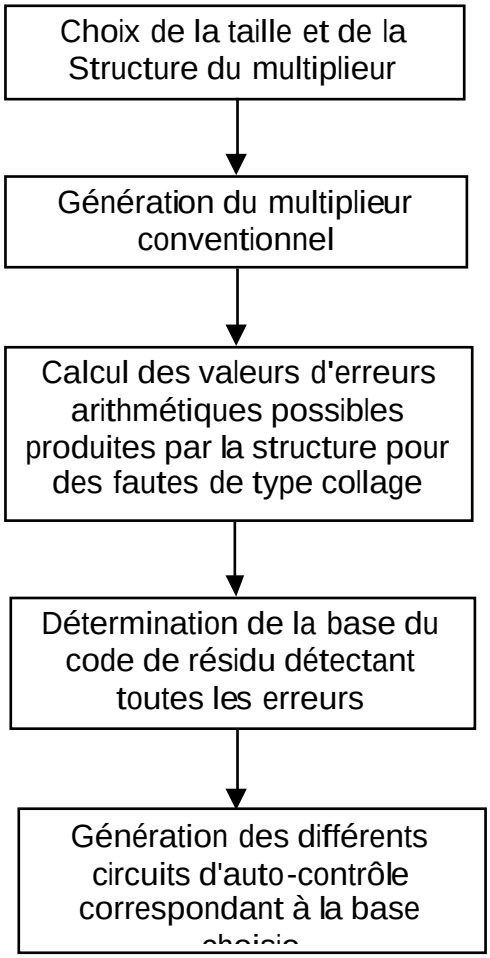


Figure 4.14. Le flot de données de nos outils de génération de multiplieurs auto-contrôlables

Chapitre 5. Opérateurs Arithmétiques Auto-Contrôlables Basés sur la Prédiction de Parité

5.1. Introduction

Le code de parité est un code qui détecte toutes les erreurs de multiplicité impaire mais pas celles de multiplicité paire. Dans le but d'obtenir des circuits sûrs en présence de fautes, il n'est donc généralement pas possible d'utiliser le code de parité pour des circuits ayant un degré de divergence maximal > à un. Un circuit dont la divergence est égale à un est un circuit où chaque nœud est connecté à une seule sortie du circuit. Dans ce cas, une erreur interne simple va produire une erreur de sortie simple. Généralement, c'est le cas des registres et des mémoires ROM et RAM. Pour les circuits de type opérateurs arithmétiques, une faute interne simple peut générer des erreurs de sortie dont la multiplicité est aléatoire, et par conséquent n'est pas toujours détectable par le code de parité.

De récents développements [Nic93] [Nic97a] [Nic97b] [Dua97a] [Dua97b] sur la conception d'opérateurs arithmétiques ont démontré la faisabilité de solutions sûres en présence de fautes basées sur la prédiction de parité, et ceci pour plusieurs blocs de base utilisés dans les chemins de données comme additionneurs, registres à décalage, ALUs et multiplieurs. Ces solutions sont très intéressantes puisque dans la plupart des cas, elles introduisent de faibles surcoûts.

Dans ce chapitre nous allons rappeler les principaux résultats obtenus dans [Nic93] [Nic97a] [Nic97b] [Dua97a] [Dua97b], et ensuite nous allons généraliser les solutions décrites pour passer de l’utilisation d’un seul bit de parité a l’utilisation de plusieurs bits de parité. Ceci permet de donner une meilleure couverture de fautes dans le cas de fautes multiples de type collage ou de fautes transitoires.

5.2. Conception d'Additionneurs et de Multiplieurs Sûrs en Présence de Fautes

Les cellules de HA et de FA sont des éléments de base dans la conception d'opérateurs arithmétiques. L'étude réalisée dans [Nic93] [Nic97a] [Nic97b] porte sur l'analyse de la propagation d'erreur dans des réseaux constitués de telles cellules. La structure générale d'un opérateur arithmétique sûr en présence de fautes basé sur la prédiction de parité est représentée par la figure 5.1. L'équation de parité est obtenue en appliquant l'opérateur de

parité sur l'équation représentant l'opération arithmétique. La duplication des retenues est expliquée par la suite.

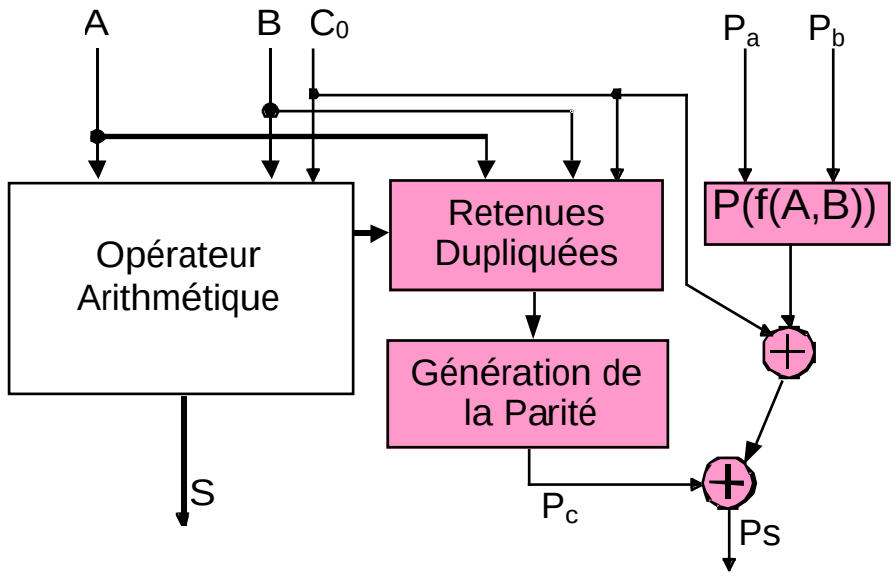


Figure 5.1. Structure générale d'un opérateur arithmétique basé sur la prédiction de parité

Dans [Nic93], on démontre que les cellules de HA et de FA, utilisées pour la conception d'opérateurs arithmétiques sûrs en présence de fautes et basés sur la prédiction de parité, doivent avoir le schéma symbolique présenté dans la figure 5.2.

Le schéma de la figure 5.2 a été obtenu à partir des contraintes suivantes :

Contrainte 1 : L'implémentation du circuit de prédiction de parité par un circuit indépendant, détruit la sûreté en présence de fautes.

Cette contrainte est évidente, puisque une faute dans l'opérateur arithmétique peut produire des erreurs de multiplicité aléatoire, y compris des erreurs de multiplicités paires. Si le circuit de prédiction de parité est indépendant, il ne sera pas affecté par ces erreurs et par conséquent elles échappent à la détection.

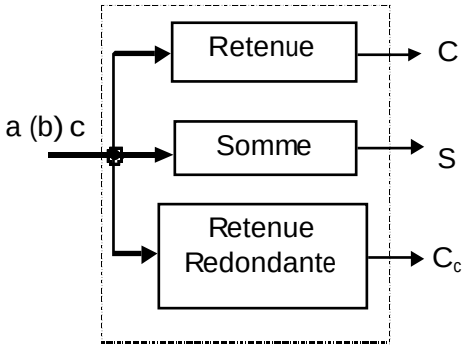


Figure 5.2. Cellule de HA ou de FA avec une retenue
indépendante pour la prédiction de parité

Contrainte 2 : La prédiction de parité à partir des retenues normales détruit la sûreté en présence de fautes.

La cellule FA/HA a les sorties retenue et somme. Soit une faute affectant la retenue seulement. Cette faute produit une erreur qui va se propager vers les sorties de l'opérateur via les signaux somme des cellules suivantes (fonction xor). Si les opérandes d'entrée sont telles que les retenues des cellules suivantes ne soient pas affectées (les entrées de ces cellules sont 00 ou 11), il y aura seulement une retenue et une sortie primaire erronées. Comme la parité est calculée à partir des retenues normales, elle sera incorrecte aussi et l'erreur simple sur la sortie de l'opérateur n'est pas détectée.

Pour la prédiction de parité ce problème est résolu en ajoutant une retenue redondante et indépendante des sorties somme et retenue de la cellule FA/HA.

Contrainte 3 : Si les sorties retenue et somme de la cellule d'addition partagent des portes logiques, l'opérateur arithmétique résultant n'est pas sûr en présence de fautes.

Soit une faute générant une erreur sur les sorties retenue et somme en même temps. L'erreur sur la sortie somme se propage vers une sortie primaire de l'opérateur. Si les opérandes de l'opérateur sont telles que les retenues des cellules conséquentes ne soient pas affectées, on aura seulement deux sorties primaires erronées. Dans ce cas les retenues redondantes ne sont pas affectées et la parité prédite est correcte alors que deux sorties primaires sont incorrectes.

Supposons maintenant que la retenue et la somme partagent des portes logiques, mais aussi que si une faute affecte ces deux sorties elle affectera en même temps la sortie retenue redondante. Dans ce cas, en se basant sur les mêmes conditions précédentes, la retenue redondante de la cellule affectée sera aussi erronée, et par conséquent la parité prédite change

de valeur alors que deux sorties primaires sont erronées. Dans ce cas l'erreur est détectée et la sûreté en présence de fautes est préservée.

Contrainte 4 : Si une branche d'entrée de la cellule d'addition est connectée à deux sorties seulement des trois sorties de la cellule, l'opérateur arithmétique perd sa sûreté en présence de fautes.

En procédant de la même manière que précédemment, on peut facilement démontrer cette contrainte.

Les figures 5.3, 5.4 et 5.5 représentent des implémentations standards des cellules d'addition tenant compte des contraintes précédentes.

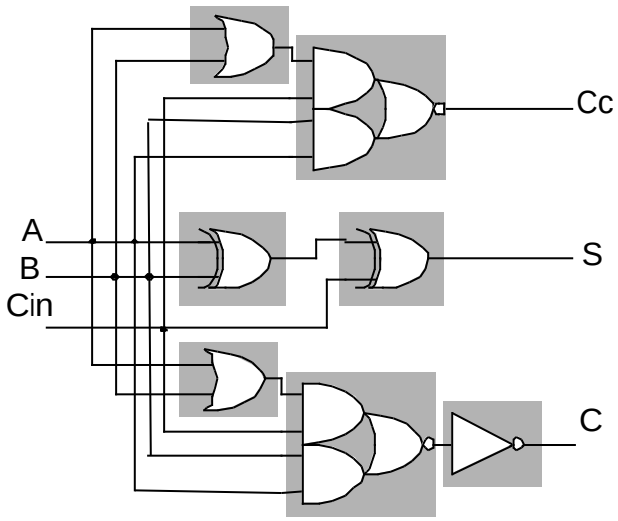


Figure 5.3. Cellule de FA avec retenue redondante et sans porte logique commune entre les différentes sorties

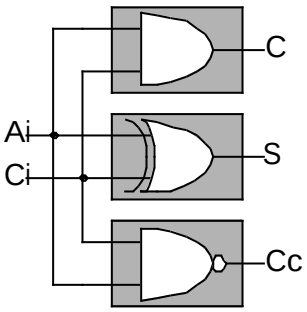


Figure 5.4. Une cellule de HA avec retenue redondante

Si une faute simple affecte une des trois cellules présentées dans les figures précédentes, soit une seule sortie est erronée, soit les trois sorties sont erronées en même temps.

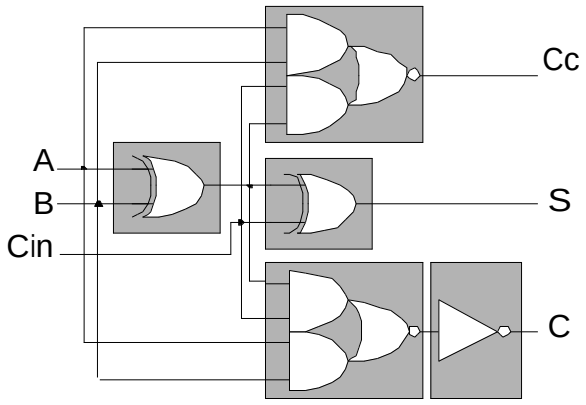


Figure 5.5. Cellule de FA avec retenue redondante et une porte logique commune entre les différentes sorties

5.2.1. Réseaux d'Addition Sûrs en Présence de Fautes Basés sur des Cellules de HA et de FA

Un réseau de cellules de HA et de FA est dit à sortance impaire si et seulement si chacune des sorties de ses cellules constitue une entrée pour un nombre impair d'autres cellules du réseau. Dans [Nic93], on démontre le théorème suivant :

Théorème : Un opérateur arithmétique basé sur les cellules de HA et FA présentées et dont la sortance est impaire est sûr en présence de fautes.
Pour démontrer ce théorème, il suffit en fait de prouver pour chaque cas de figure de faute possible que si le nombre de sorties primaires erronées est pair, alors le nombre de retenues redondantes erronées est impair et vice versa. Une faute sera donc toujours détectée et la sûreté en présence de fautes préservée.

5.3. Développements

Les solutions sûres en présence de fautes présentées dans [Nic93] [Nic97a] [Nic97b] considèrent la génération d'un seul bit de parité permettant la détection d'éventuelles erreurs. Ces solutions permettent d'assurer la propriété sûr-en-présence-de-fautes pour les fautes simples du type collage logique. Pour des fautes multiples ou pour des fautes transitoires, cette propriété n'est pas garantie. Il arrive même que la couverture soit très faible pour de telles fautes. Dans le but d'augmenter cette couverture de fautes nous avons généralisé les techniques sûres en présence de fautes utilisant un seul bit de parité en passant à l'utilisation de plusieurs bits de parité. On crée en fait plusieurs groupes de sorties tels que les bits de sortie successifs appartiennent à des groupes différents. On génère ensuite de bit de parité pour chaque groupe. Par exemple, pour deux bits de parité, on génère un bit de parité pour les

sorties d'ordre pair et un bit de parité pour les sorties d'ordre impair. La figure suivante montre le cas de trois bits de parité.

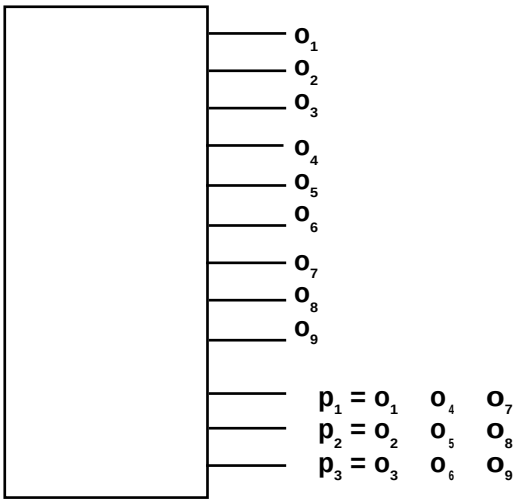


Figure 5.6. Codage à 3 bits de parité

De manière générale, soit $A_{n-1}\ A_{n-2}$ ___ $A_1\ A_0$ la représentation binaire de l'opérande *A*, On définit *g* groupes de parité de la manière suivante :

$P_{A(k)} = A_k \quad A_{k+g} \quad ??? \quad A_{k+n_k \cdot g}$

Avec $0 \le k < g$ et $k + n_k \_ g \le n-1$.

5.3.1. Additionneurs avec plusieurs Bits de Parité

Soit $S = A + B$. Dans le cas d'un seul bit de parité la relation suivante est évidente :

$$P_S = P_A \oplus P_B \oplus \textit{Parité des retenues}.$$

Pour *g* groupes de parité, la relation de parité reste facile à trouver :

$$P_{S(k)} = P_{A(k)} \quad P_{B(k)} \quad P_{C[(k-1) \bmod g]}$$

où P_C est la parité (ou les parités) des retenues.

Le cas de l'additionneur séquentiel est évident et ne sera pas développé ici. L'additionneur rapide, comme on l'a vu, n'est pas basé sur des cellules de HA et de FA. Pour garantir la sûreté en présence de fautes dans ce cas, le bloc de génération des retenues peut être dupliqué. Les sorties des deux blocs peuvent ensuite être comparées en utilisant un contrôleur double-rail pour détecter d'éventuelles erreurs dans le bloc de génération des retenues. Cette solution coûte cher puisque les dimensions du bloc de génération des retenues sont importantes par rapport aux dimensions de l'additionneur tout entier.

Une meilleure solution a été proposée dans [Nic93]. Elle consiste à générer des retenues de contrôle à partir des retenues normales de l'additionneur en utilisant la relation suivante :

$C_i' = A_i\_B_i + P_i\_C_{i-1}$ (voir figure 5.7).

Les termes C_{i-1} de la relation précédente sont les retenues normales de l'additionneur et les termes C_i' sont les retenues de contrôle. Un contrôleur double-rail permet ensuite de détecter d'éventuelles erreurs. On peut démontrer facilement que cette solution est sûre en présence de fautes. Dans [Nic93] on montre qu'en plus de sa fonction de contrôle, le contrôleur double-rail peut être utilisé pour générer le bit de parité des retenues. Pour exploiter cette propriété dans le cas de plusieurs groupes de parité on utilise un contrôleur double-rail pour chaque groupe de retenues. Ces contrôleurs génèrent les parités $P_{C(k)}$ des différents groupes de retenues. Ensuite les sorties des contrôleurs sont compressées en utilisant un dernier contrôleur double-rail pour générer un signal d'erreur pour le bloc de génération des retenues.

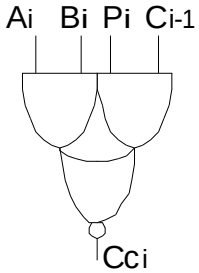


Figure 5.7. Génération de retenues d'auto-contrôle
à partir des retenues normales

La figure 5.8 représente le schéma général d'un additionneur rapide sûr en présence de fautes basé sur la solution décrite ci-dessus. Cette solution introduit un faible surcoût par rapport à la duplication.

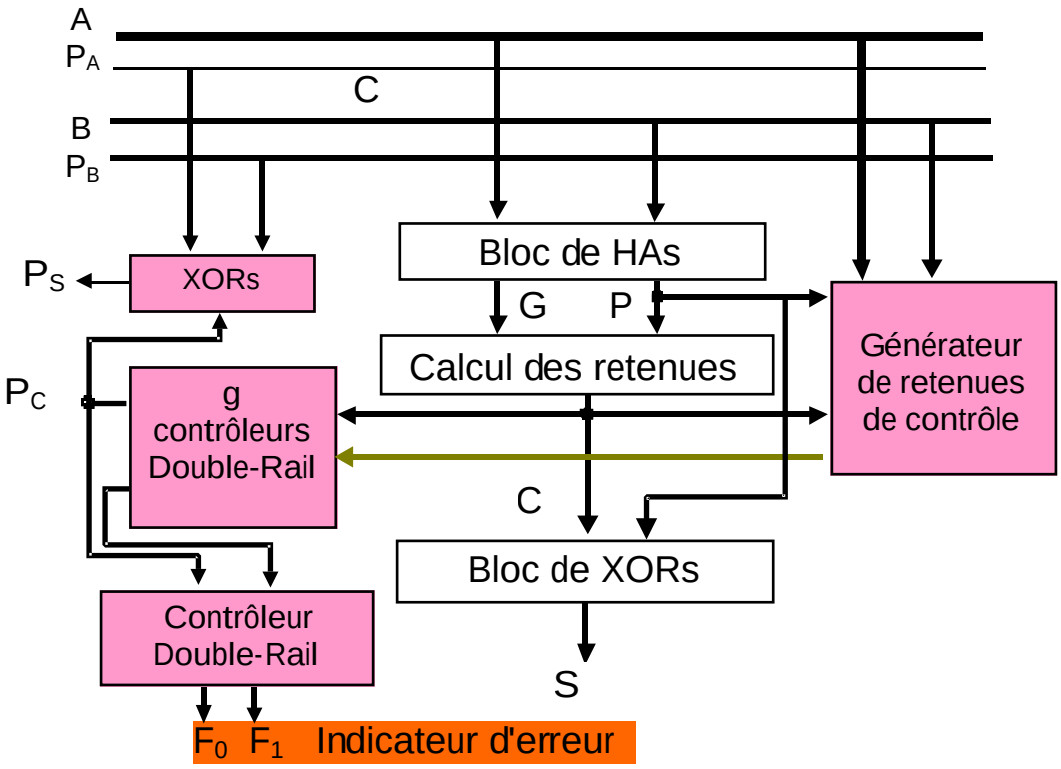


Figure 5.8. Additionneur rapide sûr en présence de fautes avec plusieurs groupes de parité

Implémentation et Résultats

Un générateur de bloc paramétrable générant la structure de la figure 5.8 est implémenté pour différentes structures d'additionneurs. En utilisant les cellules standards de la technologie ES2-CMOS-1 µm on obtient les résultats suivants dans le cas de l'additionneur de Kogge-Stone.

	8 bits	16	32	64
Simple (mm^2)	0,0670	0,1522	0,3465	0,7781
Surcoût pour 1 bit de parité	25,8	28,8	21,6	23,1
Surcoût pour 2 bits de parité	26,9	29,2	21,8	23,2
Surcoût pour 3 bits de parité	28,1	29,8	22,0	23,3

Table 5.1. Surcoût de l'additionneur de Kogge-Stone basé sur la prédiction de parité

Dans la table 5.1 on peut vérifier que les coûts en surface de l'additionneur sûr en présence de fautes sont pratiquement les mêmes pour 1, 2 et 3 bits de parité.

5.3.2. Multiplieurs Avec Plusieurs Bits de Parité

On a vu que les multiplieurs utilisés en pratique, et présentés dans les chapitres précédents, sont implémentés en utilisant un circuit qui génère les produits partiels et un réseau de FAs et de HAs pour additionner ces produits partiels. La sortie somme d'une cellule FA ou HA est égale à la somme modulo 2 des entrées et donc elle représente la parité des entrées de la cellule. Par conséquent, le relation suivante est triviale

la parité du produit = la parité des produits partiels ⊕ la parité des retenues générées par les cellules de FA et de HA

Ou : $P_{OUT} = P_{PP} \oplus P_C$.

5.3.2.1. Opérandes Non Codées

Si les opérandes ne sont pas codées, le produit partiel PP_{ij} est égal à $a_j \wedge b_j$. On peut donc démontrer facilement que la somme modulo 2 des termes $a_i \wedge b_j$ est égale à $P_A \wedge P_B$ et par suite :

$P_{OUT} = (P_A \wedge P_B) \oplus P_C$.

Où P_A et P_B sont les parités des opérandes d'entrée et P_C est la parité des retenues internes.

Dans le cas où la parité est prise en plusieurs groupes, on calcule d'abord les différentes parités des groupes de produits partiels. Les différentes colonnes de produits partiels dans le multiplieur sont représentées par les lignes suivantes :

0	a_0b_0
1	$a_1b_0 + a_0b_1$
2	$a_1b_1 + a_2b_0 + a_0b_2$
3	_ _ _
_ _ _	
_ _ _	
2n-3	$a_{n-2}b_{n-1} + a_{n-1}b_{n-2}$
2n-2	$a_{n-1}b_{n-1}$

Soit g le nombre des groupes de parité et PP_i $(0 \leq i < g)$ la parité i. PP_i est la somme modulo 2 des lignes i, $i+g$, $i+2g$, ___.

A partir des lignes de produits partiels précédentes on peut écrire :

$PP_i = a_0 (b_i \oplus b_{i+g} \oplus b_{i+2g} \oplus \_\_\_) \oplus a_1(b_{i+g-1} \oplus b_{i+2g-1} \oplus b_{i+3g-1} \oplus \_\_\_) \oplus$

$a_k (b_{i+g-k} \oplus b_{i+2g-k} \oplus b_{i+3g-k} \oplus \_\_\_) \oplus \_\_\_ \oplus a_{n-1} (b_{i+g-(n-1)} \oplus b_{i+2g-(n-1)} \oplus b_{i+3g-(n-1)} \oplus \_\_\_)$

avec $0 \leq i + q\_g - k \leq n-1$.

En remarquant que les termes a_m, a_{m+g}, a_{m+2g}, ___ ($m < g$) sont multipliés par la même valeur, PP_i peut s'écrire sous la forme suivante :

$PP_i = (a_0 \oplus a_g \oplus a_{2g} \oplus \_\_\_)\_(b_i \oplus b_{i+g} \oplus b_{i+2g} \oplus \_\_\_) \oplus$

$(a_1 \oplus a_{g+1} \oplus a_{2g+1} \oplus \_\_\_)\_(b_{i+g-1} \oplus b_{i+2g-1} \oplus b_{i+3g-1} \oplus \_\_\_) \oplus \_\_\_$

$\oplus (a_{g-1} \oplus a_{2(g-1)} \oplus a_{3(g-1)} \oplus \_\_\_)\_(b_{i+1} \oplus b_{i+1+g} \oplus b_{i+1+2g} \oplus \_\_\_)$

qui est équivalente à :

$$PP_i = \prod_{k=0}^{g-1} \left(P_{A(k)} \; ? P_{B(i-k) \bmod g} \right)$$

Les valeurs des parités du produit sont donc données par :

$$P_{out(i)} = \prod_{k=0}^{g-1} \left(P_{A(k)} \; ? P_{B[(i-k) \bmod g]} \right) \quad P_{C(i)}$$

Où $P_{C(i)}$ est la parité des retenues additionnées au groupe de produits partiels numéro i.

La figure 5.9 représente un multiplieur de Wallace basé sur la prédiction de parité avec un ou plusieurs bits de parité. La partie de contrôle concernant l'additionneur est identique à celle de la section précédente et les HA/FA utilisés sont ceux des figures 5.3, 5.4 et 5.5.

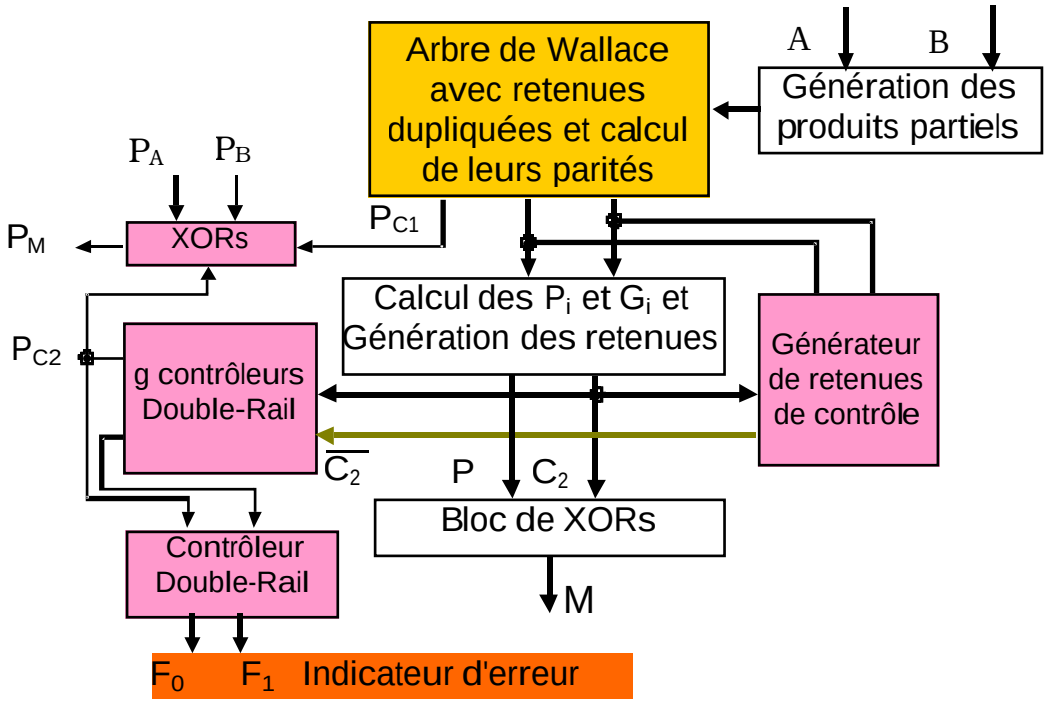


Figure 5.9. Multiplieur de Wallace basé sur la prédiction de parité avec un ou plusieurs bits de parité

Implémentation et Résultats

Un générateur de bloc paramétrable générant la structure de la figure 5.9 est implémenté pour différentes structures d'additionneurs. En utilisant les cellules standards de la technologie ES2-CMOS-1 μm et le schéma de FA de la figure 5.5 (type 2) on obtient les résultats suivants dans le cas de l'additionneur CLA.

	8x8 bits	16x16	32x32
Simple (mm^2)	0,3082	1,2405	4,9412
Surcoût pour 1 bit de parité (%)	42,8	44,4	45,6
Surcoût pour 2 bits de parité (%)	41,2	44,4	45,6
Surcoût pour 3 bits de parité (%)	41,0	44,3	45,5

Table 5.2. Suroûts du multiplieur Wallace-CLA basé sur la prédiction de parité

La table 5.2 montre bien que l'augmentation du nombre des bits de parité n'introduit pas de coût supplémentaire.

5.3.2.2. Opérandes Codées

Dans le cas des multiplieurs de Booth, on trouve aussi le bloc qui génère les produits partiels et le réseau de cellules d'addition. La relation $P_{OUT} = P_{PP} \oplus P_C$ est donc toujours valable, mais le calcul de P_{PP} est plus complexe vu le codage des opérandes.

Une faute dans le circuit de codage détruit la sûreté en présence de fautes. Ce problème est résolu en dupliquant le circuit de codage et en testant ses sorties au moyen d'un contrôleur double-rail.

Pour calculer les parités des produits partiels, on se réfère à la figure 5.10. Cette figure montre un exemple de multiplication de Booth de deux nombres signés de 8 bits chacun. Le codage utilisé est le codage compact (voir chapitre 4) et la méthode de génération de signe est appliquée.

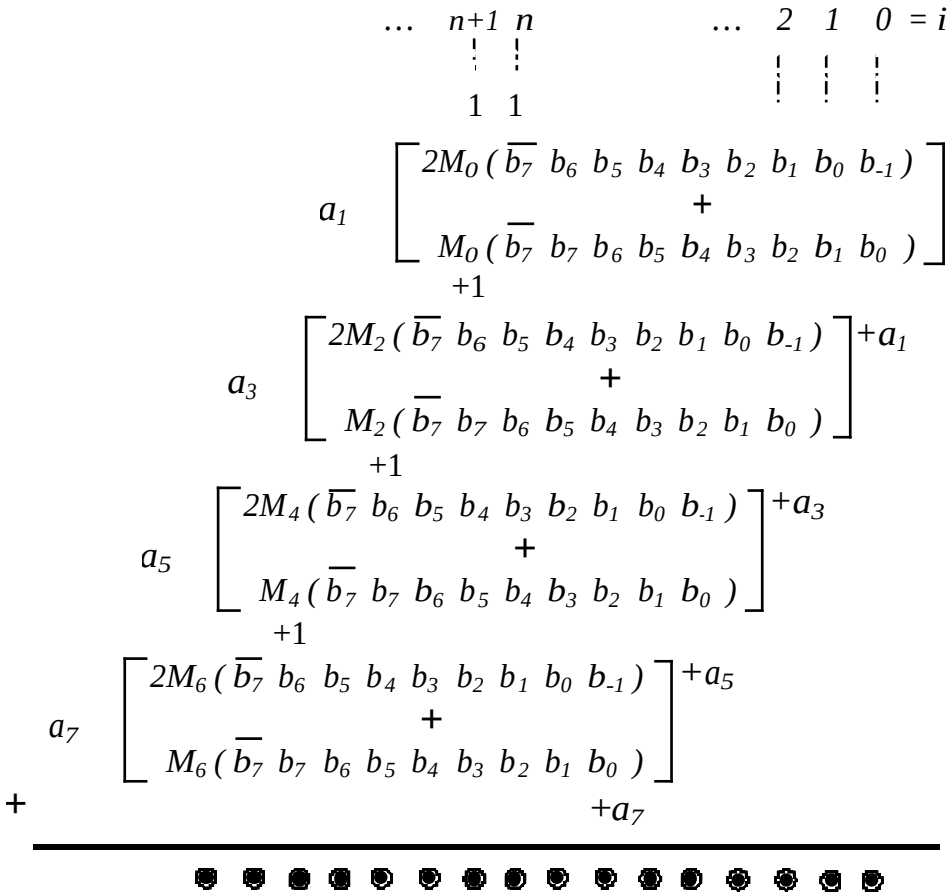


Figure 5.10. Exemple de multiplication de Booth utilisant le codage compact et la méthode de génération de signe

Pour chacun des produits partiels de la figure 5.10, le couple (M_k , $2M_k$) peut prendre seulement les valeurs (*0, 0*), (*0, 1*), et (*1, 0*). Ceci nous conduit à remplacer les *OUs* logiques par des *XOR's*. En utilisant la figure 5.10, on obtient donc la relation suivante :

$$PP_i = \bigoplus_{k=0} \left(M_{2k} \cdot P_{B|i-2k|_g}\right) \left(2M_{2k} \cdot P_{B|i-2k-1|_g}\right) \left|\frac{n+g-|i-2k|_g}{g}\right|_2 \cdot a_{2k+1} \cdot \left[\left(|i-2k|_g = 0\right) \cdot a_{2k+1}\right] \left[\left(|n+2k|_g = i\right) \cdot \overline{M_{2k} b_{n-1}}\right] \left[\left(|n+2k+1|_g = i\right)\right] \cdot \left[\left(|n|_g = i\right)\right]$$

Où :

- *g* est le nombre des groupes de parité.
- *K* est le numéro de ligne du produit partiel (on commence par 0).
- $\frac{n+g-|i-2k|_g}{g}$ est le nombre de a_{2k+1} existant dans la ligne *k*.

- $(|n+2k|_g = i) ? \overline{M_{2k} b_{n-1}}$ existe si et seulement si $i = |n + 2k|_g$ (i.e. le bit de signe du produit partiel numéro *k* est compris dans le groupe de parité numéro *i*).
- $(|i-2k|_g = 0) . a_{2k+1}$ existe si et seulement si $|i - 2k|_g = 0$. Cette condition est vérifiée lorsque a_{2k+1}, qui est ajouté au premier bit du produit partiel numéro *k*, est compris dans le groupe de parité numéro *i*.
- $(|n+2k+1|_g = i)$ existe si et seulement si $|n+2k+1|_g = i$ et donne la valeur 1 qui veut dire une inversion sur la parité numéro *i*.
- $(|n|_g = i)$ existe si et seulement si $(|n|_g = i)$ et donne la valeur 1 qui veut dire une inversion sur la parité numéro *i*.

A titre d'exemple, les parités sont calculées pour $n = 8$, $g = 2$ et g = 3.

n = 8 et g = 2 :

$PP_0 = (P_{B(0)} \quad b_7) ? (\quad M_{2k}) \quad P_{B(1)} ? (\quad 2M_{2k}) \quad 1$

$PP_1 = P_{B(1)} ? (\quad M_{2k}) \quad P_{B(0)} ? (\quad 2M_{2k})$

n = 8 et g = 2 :

$PP_0 = (M_0 ? P_{B(0)} \quad M_2 ? P_{B(1)} \quad M_4 ? P_{B(2)} \quad M_6 ? P_{B(0)})$
$(2M_0 ? P_{B(2)} \quad 2M_2 ? P_{B(0)} \quad 2M_4 ? P_{B(1)} \quad 2M_6 ? P_{B(2)}) \quad a_3 \quad a_5 \quad \overline{M_4 ? b_7}$

$PP_1 = (M_0 ? P_{B(1)} \quad M_2 ? P_{B(2)} \quad M_4 ? P_{B(0)} \quad M_6 ? P_{B(1)})$
$(2M_0 ? P_{B(0)} \quad 2M_2 ? P_{B(1)} \quad 2M_4 ? P_{B(2)} \quad 2M_6 ? P_{B(0)}) \quad a_1 \quad a_3 \quad a_7 \quad M_2 ? b_7$

$PP_2 = (M_0 ? P_{B(2)} \quad M_2 ? P_{B(0)} \quad M_4 ? P_{B(1)} \quad M_6 ? P_{B(2)})$
$(2M_0 ? P_{B(1)} \quad 2M_2 ? P_{B(2)} \quad 2M_4 ? P_{B(0)} \quad 2M_6 ? P_{B(1)}) \quad a_1 \quad a_5 \quad a_7 \quad b_7 ? (M_0 \quad M_6)$

Sur la relation de parité des produits partiels on peut remarquer les deux points suivants :

- Pour une PP_i, les M_{2k}, M_{2k+2g}, ___ sont multipliés par la même valeur de parité $P_{B|i-2k|_g}$.
- Pour une PP_i, les $2M_{2k}$, $2M_{2k+2g}$, ___ sont multipliés par la même valeur de parité $P_{B|i-2k-1|_g}$.

Ceci permet de faire les simplifications suivantes :

$P_{B|i-2k|_g} ? M_{2k} \quad P_{B|i-2k|_g} ? M_{2k+2g} \quad ??? = P_{B|i-2k|_g} ? (M_{2k} \quad M_{2k+2g} \quad ???)$

$P_{B\lfloor i-2k-1\rfloor_g}$?$2M_{2k}$ $P_{B\lfloor i-2k-1\rfloor_g}$?$2M_{2k+2g}$???= $P_{B\lfloor i-2k-1\rfloor_g}$?$(2M_{2k}$ $2M_{2k+2g}$???$)$

Les valeurs entre les parenthèses représentent les parités des sorties du bloc de codage. Pour calculer ces parités on peut exploiter les propriétés du contrôleur double-rail qui vérifie les sorties du circuit de codage. On implémente ce contrôleur en utilisant deux groupe de g contrôleurs double-rail chacun. Le premier groupe vérifie les M_{2k} et génère leurs parités, le deuxième fait de même pour les signaux $2M_{2k}$. Les sorties des groupes de contrôleurs sont ensuite compressées pour générer le signal d'erreur. La figure 5.11 montre cette implémentation. Cette technique permet de réduire considérablement le surcoût.

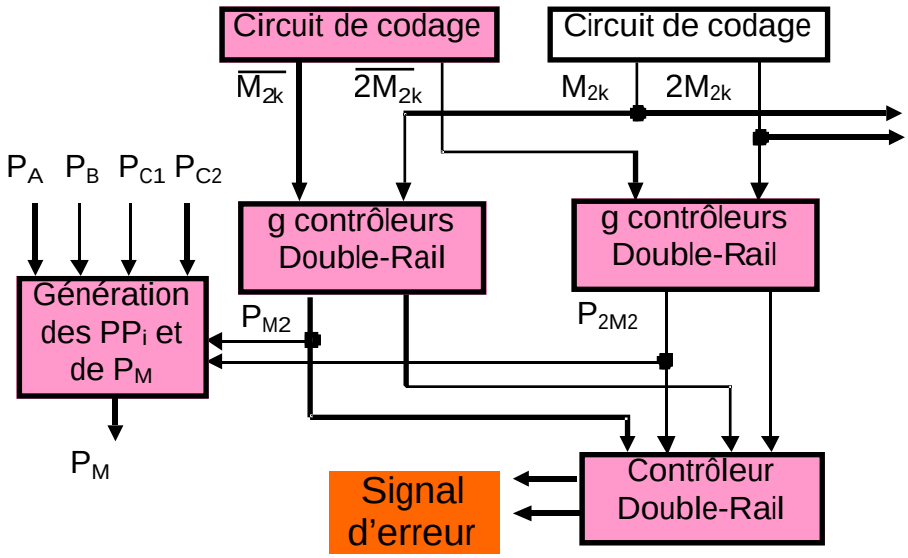


Figure 5.11. Exploitation du contrôleur double-rail pour calculer les parités

Implémentation et Résultats

Un générateur de bloc paramétrable générant la structure de la figure 5.11 est implémenté pour différentes structures d'additionneurs. En utilisant les cellules standards de la technologie ES2-CMOS-1 µm on obtient les résultats de la table 5.3 dans le cas de l'additionneur de Kogge-Stone.

	8x8 bits	16x16	32x32
Simple (mm^2)	0,328	1,107	3,911
Surcoût pour 1 bit de parité (%)	39,3	34,9	34,9
Surcoût pour 2 bits de parité (%)	40,8	35,3	35,1

Surcoût pour 3 bits de parité (%)	54,2	39,2	36,8

Table 5.3. Surcoûts du multiplieur Booth Wallace-Kogge-Stone basé sur la prédiction de parité

En comparaison avec les multiplieurs à opérandes non codées (section précédente), le surcoût augmente légèrement pour trois bits de parité. Ceci est dû à la complexité plus importante du circuit de prédiction de parité dans le cas du codage de Booth, qui dépend du nombre de bits de parité d'une part (pair ou impair) et de la taille des opérandes d'autre part.

5.4. Registres à Décalage Auto-Contrôlables Basés sur le Code De Parité

5.4.1. Introduction

Les registres à décalage sont des unités qui peuvent être utilisées pour effectuer plusieurs opérations : division par 2^k (décalage logique à droite), multiplication par 2^k (décalage logique à gauche), rotations (à gauche ou à droite), ou décalages sur des nombres signés (décalages arithmétiques). Dans ce chapitre on reprend l'étude de registres à décalage sûrs en présence de fautes réalisée dans [Dua97c]. Cette étude porte sur l'implémentation de registres à décalage utilisant des cellules standards. La solution sûre en présence de fautes proposée est basée sur la prédiction de parité et donne un surcoût très faible. Dans ce chapitre on va rappeler le principe et la solution sûre en présence de fautes des registres à décalages étudiés (basés sur des multiplexeurs), ainsi que les résultats obtenus. Ensuite, on va développer cette étude en introduisant deux bits de parité indépendants en vue d'augmenter la couverture de fautes. Au delà de deux bits de parité, le surcoût devient, comme on va le voir, élevé.

5.4.2. Registres à Décalage Basés sur des Multiplexeurs

Le schéma général du registre à décalage considéré est celui basé sur des multiplexeurs proposé dans [Hwa79]. Ce schéma est illustré dans la figure 5.1. Dans ce schéma, la valeur arithmétique des signaux $C_{log2(n)-1}$, ___, C_1, C_0 détermine le nombre de positions à décaler. Le nombre de positions à décaler est égale à $\sum_{i=0}^{\log_2(n)-1} C_i \,?2^i$. Cette équation est implémentée en cascadant $log_2(n)$ registres à décalage, dont chacun permet de décaler un nombre fixe de positions. Ainsi, le *ième* registre permet d'effectuer un décalage de zéro ou de 2^i positions. Un rang de multiplexeurs 2:1 contrôlés par le signal C_i sélectionne ensuite les sorties du *ième*

registre pour $C_i = 1$ ou les entrées du *ième* registre pour $C_i = 0$. Cette implémentation est représentée sur la figure 5.1 pour $n = 4$.

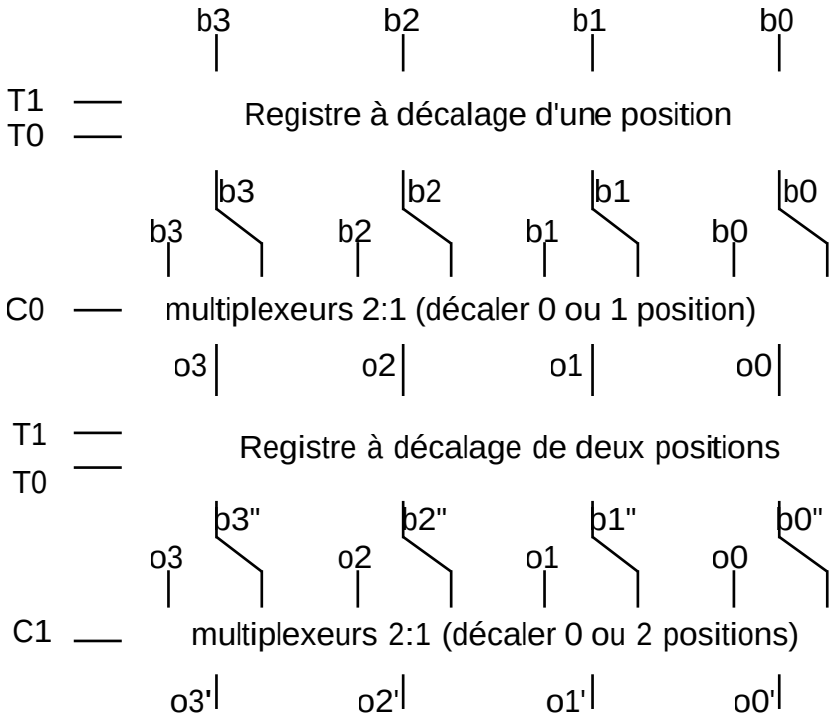


Figure 5.1. Schéma général d'un registre à décalage basé sur des multiplexeurs [Hwa79]

Un registre qui effectue un décalage de nombre fixe de positions est un circuit de routage simple. Pour un registre permettant d'effectuer plusieurs opérations, chacun des registres à décalage d'un nombre fixe de positions est contrôlé par les signaux de contrôle qui déterminent le type de l'opération à effectuer. L'exemple de la figure 5.1 montre les 4 opérations de décalage les plus usuelles (rotation, décalage logique à gauche, décalage logique à droite et décalage arithmétique). L'opération à effectuer est déterminée par les valeurs des signaux de contrôle t_0 et t_1 : $t_1t_0 = 00$ pour la rotation à droite (ROT), $t_1t_0 = 01$ pour le décalage logique à gauche (DLG), $t_1t_0 = 10$ pour le décalage logique à droite (DLD) et $t_1t_0 = 11$ pour le décalage arithmétique à droite (DAD). Dans ce cas, chacun des registres à décalage d'un nombre fixe de positions est implémenté utilisant des multiplexeurs 4:1 qui sélectionnent les signaux de sortie selon la valeur de t_1t_0. Les figures 5.2 et 5.3 représentent des registres à décalage d'une position et de deux positions respectivement pour un nombre de bits $n = 4$.

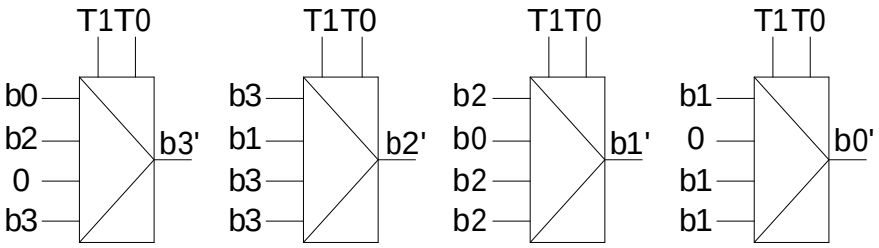


Figure 5.2. Un registre à décalage d'une position à 4 bits

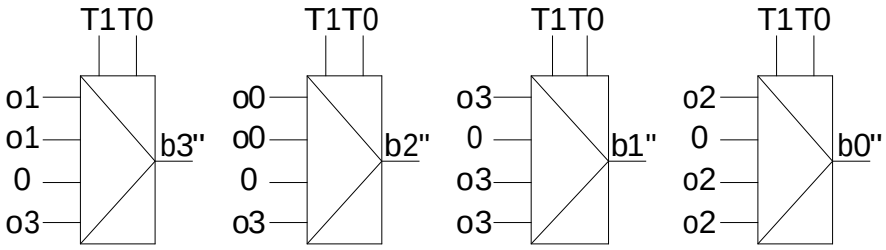


Figure 5.3. Un registre à décalage de deux positions à 4 bits

5.4.3. Solution Sûre en Présence de Fautes

La solution sûre en présence de fautes basée sur la prédiction de parité a été présentée dans [Dua97a]. Un circuit de prédiction de parité indépendant est implémenté pour chaque type d'opération de décalage. Un circuit de décodage permet ensuite de sélectionner la valeur correcte de parité parmi les valeurs de parités produites par les circuits de prédiction de parité selon les valeurs des signaux de contrôle $C_{log2(n)-1}$, ___, C_1, C_0 et T_1, T_0. Un arbre de prédiction de parité de coût minimum et de délai logarithmique est trouvé. Un exemple de cet arbre est représenté par la figure 5.4 pour $n = 16$. La parité p_k représente la parité des k bits les moins significatifs et p_k' représente la parité des k bits les plus significatifs de l'opérande d'entrée.

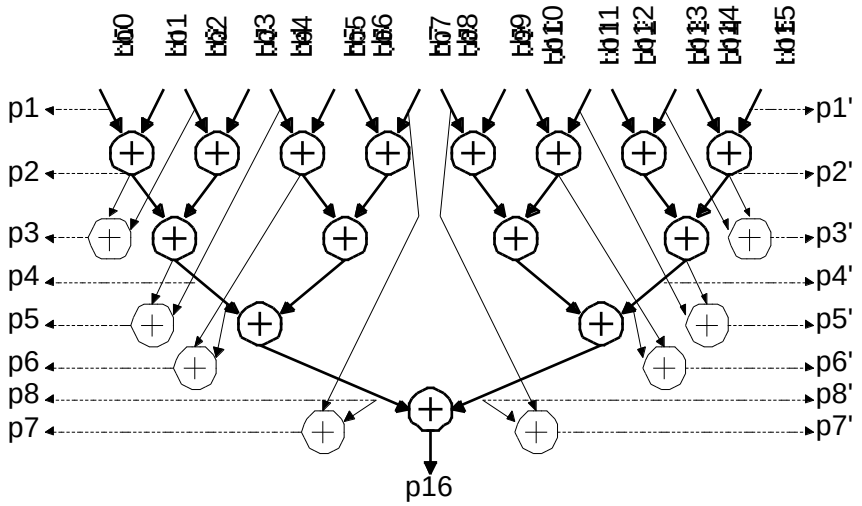


Figure 5.4. Arbre de parité logarithmique

Soit n le nombre de bits de l'opérande d'entrée (supposé pair) et soit k le nombre de positions à décaler ($1 \leq k \leq n-1$). Dans le cas du décalage logique à gauche on montre que la parité est donnée par les équations suivantes :

$$pour\ k < 2^{r-1},\ Pr = Pi \oplus p_k' \qquad Equation\ 1$$

$$pour\ k \geq 2^{r-1},\ Pr = p_{(n-k)} \qquad Equation\ 2$$

Dans les cas de décalage logique à droite et de décalage arithmétique à droite on montre aussi les équations suivantes :

$$pour\ k < 2^{r-1},\ Pr = Pi \oplus p_k \oplus d \quad \text{pour k impair} \qquad Equation\ 3$$

$$pour\ k \geq 2^{r-1},\ Pr = p_{(n-k)}' \oplus d \quad \text{pour k impair} \qquad Equation\ 4$$

où d est égale à 0 pour le décalage logique et au bit de poids le plus significatif b_{n-1} dans le cas du décalage arithmétique.

Le circuit de décodage correspondant aux équations précédentes est représenté dans la figure 5.5. Pendant la rotation, la parité est préservée; par conséquent, la parité de l'opérande d'entrée p_i est transférée à p_r. Le bloc A de la figure implémente les équations du décalage logique à gauche. Les équations du décalage logique à droite sont implémentées par le bloc B. Les équations du décalage arithmétique à droite sont les mêmes que celles du décalage logique à droite sauf que d peut être égale à 1. Les k bits les plus à gauche sont remplacés par le bit b_{n-1};

par conséquent, d est égale à (k modulo 2)_b_{n-1}. Pour k impair (C_0 = 1) on a donc d = b_{n-1} et pour k pair on a d = 0. Ceci correspond à d = C_0_b_{n-1} et est implémenté par le bloc C.

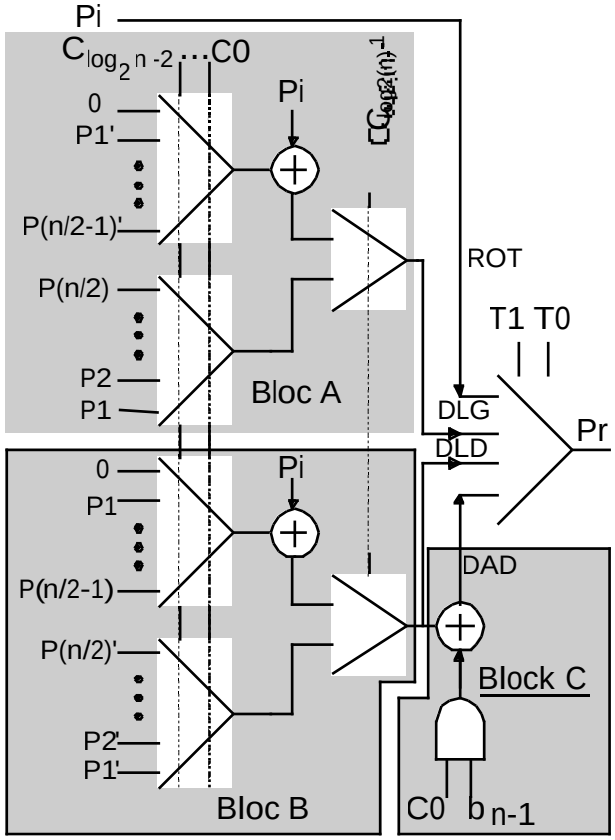


Figure 5.5. Le circuit de décodage

Le schéma du registre à décalage auto-contrôlable intégré dans un chemin de données est représenté dans la figure 5.6.

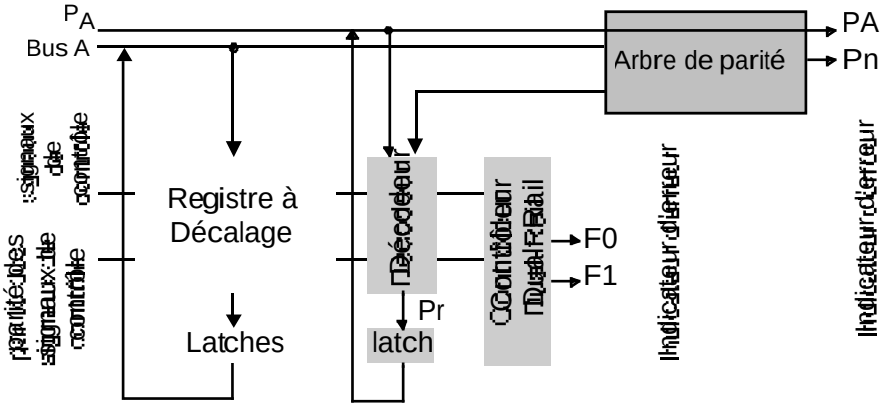


Figure 5.6. Registre à décalage auto-contrôlable

En considérant les fautes de type collage à *z* au niveau cellules standards, on montre que le schéma de la figure 5.6 est sûr en présence de fautes.

Par la suite on va généraliser la solution précédente pour un nombre quelconque de bits de parité, en vue d'augmenter la couverture de fautes. Ensuite, on va réaliser une implémentation pour deux bits de parité et comparer le surcoût par rapport à un seul bit de parité.

5.4.4. Registre à Décalage avec plusieurs Bits de Parité

La solution pour g bits de parité consiste à diviser le décodeur et l'arbre de parité présentés précédemment en g décodeurs et g arbres de parité. L'arbre de parité d'indice i calcule les différentes parités du groupe de parité d'indice i et reçoit à ses entrées les bits de l'opérande d'entrée correspondant au groupe de parité d'indice i. Le décodeur d'indice i génère la parité du groupe de parité d'indice i selon les signaux de contrôle. Les parités de sortie sont ensuite calculées à partir des parités générées par les décodeurs en utilisant des blocs de multiplexage qui tiennent compte du type de l'opération de décalage et du nombre de positions décalées.

Soit n le nombre de bits de l'opérande d'entrée (qu'on suppose pair), k le nombre de positions à décaler et q l'indice des g groupes de parités ($0 \le q < g$). En adoptant les mêmes notations que pour un seul bit de parité, on peut calculer les équations de parité pour g groupes de parité. Ces équations ont les mêmes formes que celles déjà trouvées et sont présentées par la suite.

Décalage logique à gauche :

pour $k < n/2$ on a

$$\Pr(q) = Pi[(q-k) \bmod g] \quad Pi'[(q-k) \bmod g]_{[a-b]}$$

où :

$a = \dfrac{n-1-(q-k) \bmod g}{g} + 1$ est le nombre de bits correspondant au groupe [(q-k) *mod* g] se trouvant dans l'opérande d'entrée avant décalage.

$b = \dfrac{n-1-k-(q-k) \bmod g}{g} + 1$ est le nombre de bits correspondant au groupe [(q-k) *mod* g] se trouvant dans l'opérande d'entrée après décalage de k positions.

pour $k \ ? \ n/2$ on a

$$\Pr(q) = Pi[(q-k) \bmod g]_b$$

Décalage logique et arithmétique à droite :

pour $k < n/2$ on a :

$$\Pr(q) = Pi[(q+k) \bmod g] \quad Pi[(q+k) \bmod g]_{\frac{k-1+g-(q+k) \bmod g}{g}} \qquad ((c-d) \bmod 2) ? d$$

et pour $k ? n/2$ on a :

$$\Pr(q) = Pi'[(q+k) \bmod g]_{\frac{n-1-(q+k) \bmod g}{g}} \_ \frac{k-1-(q+k) \bmod g}{g} \qquad ((c-d) \bmod 2) ? d$$

avec

$$c - d = \frac{n-1+k-(q+k) \bmod g}{g} - \frac{n-1-(q+k) \bmod g}{g}$$

Ces équations peuvent être implémentées de la même manière que dans le cas d'un seul bit de parité en réalisant un sous décodeur pour chaque groupe de parité. Chacun de ces sous décodeurs reçoit les parités de son groupe générées par son arbre de parité et produit une parité partielle selon *k* et le type d'opération effectuée. Ces parités partielles passent ensuite par des circuits de multiplexage permettant de générer les g parités de sortie selon le nombre de positions décalées (*k*). Le sous décodeur d'indice *i* ($0 \leq i < g$) est composé de quatre circuits de multiplexage implémentant les quatre équations ci-dessus pour le groupe de parité d'indice *i.*

5.4.5. Implémentation pour deux bits de parité

La figure 5.7 représente la solution décrite ci-dessus dans le cas de deux bits de parité. Les deux arbres de parité ont la même structure que la figure 5.4 et génèrent les parités partielles utilisées dans les équations présentées auparavant. Il y a aussi un décodeur pour chaque groupe de parité. Ces décodeurs ont la même structure que le décodeur de la figure 5.5. Deux multiplexeurs permettent ensuite de sélectionner les parités de sortie selon la valeur de C_0.

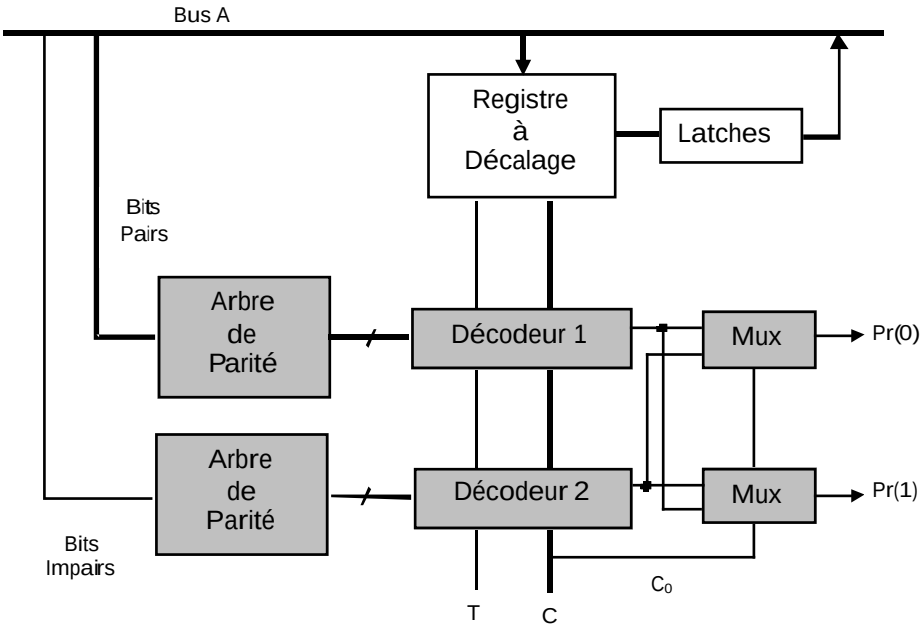


Figure 5.7. Registre à décalage avec deux bits de parité

Résultats

Nous avons implémenté des générateurs de macro-blocs paramétrables qui permettent de générer un registre à décalage basé sur la solution décrite précédemment.

En utilisant les cellules standards de la technologie ES2-CMOS-1 μm on trouve les résultats de la table 5.4.

Taille du registre	2 bits de parité		1 bit de parité	
	Coût (mm^2)	Surcoût(%)	Coût (mm^2)	Surcoût(%)
16x16	0,379	35,9	0,329	18,0
32x32	0,907	30,1	0,806	15,6
64x64	2,038	21,8	1,857	11,0

Table 5.4. Coût et surcoût pour 1 et 2 bits de parité

On peut remarquer que le surcoût pour 2 bits de parité est quasiment le double que celui pour 1 seul bit de parité. Ceci est dû au fait que la complexité des décodeurs (que l'on peut penser deux fois plus simples) utilisés pour chaque groupe de parité est comparable à celle du décodeur de la figure 5.5 (un seul bit de parité).

Contrairement aux additionneurs et aux multiplieurs, l'augmentation du nombre des bits de parité est très coûteux dans le cas des registres à décalage et ne constitue pas un bon choix.

5.5. Comportement en présence de Fautes Transitoires

Les circuits auto-contrôlables présentés dans cette thèse permettent la détection de fautes permanentes et transitoires. Le taux de couverture dans le cas de fautes permanentes du type collage logique est de 100%.

Dans le cas de fautes transitoires, tels que par exemple les SEUs (Single-Event-Upsets) produits par les impacts dans un circuit des particules ionisantes, la propriété sûr-en-présence de-fautes n'est pas assurée. Il arrive même que la couverture des erreurs reste très faible de l'ordre de 50%. Afin d'améliorer cette couverture on se propose d'utiliser des codes à plusieurs bits de parité. Les implémentations sûres en présence de fautes ainsi obtenues garantissent une couverture à 100% des erreurs produites par les fautes de type collage logique, et améliorent la couverture des erreurs dues à d'autres types de fautes tels que les fautes transitoires. Concernant les fautes de type collage logique, on sait que les solutions proposées avec un seul bit de parité sont sûres. Dans le cas de plusieurs bits de parité, pour garantir la sûreté, une faute doit être détectable par au moins un groupe de parité. Si la faute n'est détectée par aucun groupe de parité, cela veut dire qu'elle n'est pas détectée par la solution à un seul bit de parité. On peut démontrer cela en remarquant que la parité dans le cas d'un seul bit de parité égal la parité des groupes de parité dans le cas de plusieurs groupes de parité. Ceci nous conduit à affirmer aussi que la faute doit être détectée par un nombre impair des groupes de parité (la parité est réalisée avec des XORs). Dans le cas de 4 groupes de parité par exemple, une faute sera détectée par un ou trois groupes parmi les quatre groupes de parité.

Ces solutions sont intéressantes si on cherche à protéger le circuit contre tout type de fautes. Néanmoins, les fautes du type collage logique peuvent facilement être détectées et les circuits défaillants éliminés lors du test de fabrication. D'autre part les fautes transitoires, n'étant pas permanentes, ne peuvent pas être éliminées lors d'un test de fabrication. Ces fautes, de plus en plus importantes dans les technologies sous-microniques avancées, devraient devenir une cause majeure du mauvais fonctionnement dans les technologies CMOS de 0,12 à 0,1 micron et au delà.

Il est donc intéressant de protéger les circuits tout en ignorant les fautes de type collage logique. Ainsi, les contraintes imposées pour assurer la sûreté en présence de fautes pour les collages, peuvent être levées. En fait, dans ce contexte, une analyse théorique [Ang 2000] montre que :

- L'utilisation d'un bit de parité donne une couverture insuffisante face aux erreurs dues à des fautes transitoires.
- La duplication des retenues est très insuffisante face aux fautes transitoires, malgré son coût significatif.
- L'augmentation du nombre des bits de parité peut apporter une amélioration très significative de la couverture des erreurs.

A partir de ces remarques, on a été amené à modifier nos outils afin de générer des opérateurs arithmétiques avec plusieurs bits de parié mais sans duplication des retenues. L'élimination de la duplication des retenues a permis de réduire le coût matériel de façon substantielle.
Les outils ont été utilisés ensuite pour synthétiser des opérateurs dont l'efficacité de couverture d'erreurs a été évaluée par simulations de fautes transitoires. Plus précisément, les simulations sont réalisées en injectant des impulsions de largeur égale à *0,6 ns* (cas typique des transitions dues à des particules physiques) et les circuits simulés sont tous de taille 16X16. Les résultats sont présentés dans le tableau 5.5. On observe une nette augmentation de la couverture d'erreur et une diminution du coût matériel.

	Sûr 1 bit de parité		Non sûr 1 bit de parité		Non sûr 2 bits de parité		Non sûr 4 bits de parité	
Additionneur	Effic.(%)	Coût(%)	Effic.(%)	Coût(%)	Effic.(%)	Coût(%)	Effic.(%)	Coût(%)
Brent-Kung	83,7	28,3	82,1	8,2	85,2	13,1	93,8	21,3
CLA	84	29,9	81,5	9,2	84,3	14,9	92,8	23,2
Han-Carlson	82,9	27,9	81,9	7,9	83,2	12,7	93,5	20,3
Kogge-Stone	81,5	25,8	80,2	6,8	82,9	10,7	93,2	17,5
Sklansky	81,8	26,7	81,3	6,7	83,5	11,6	92	18,7
Mult. Wallace	76,9	32,2	72,5	16,1	77,4	18,3	80,1	19,4

Tableau 5.5. Surcoût et efficacité de différentes versions d'additionneurs et du multiplieur de Wallace.

Comme on peut le voir dans le tableau, les circuits sûrs en présence de fautes ne donnent pas une couverture de 100% face à des fautes transitoires. Les versions non sûres ont un coût moins important et dans le cas des additionneurs, ils donnent une meilleure couverture à partir de deux bits de parité. Dans le cas du multiplieur, la couverture est meilleure pour 4 bits de parité.

Chapitre 6. L'Outil de CAO pour la Génération de Circuits Auto-Contrôlables

L'outil de CAO utilisé pour générer les circuits étudiés dans cette thèse a été développé par Duarte [Dua97c] et Pederson [Ped95] dans le but d'automatiser la génération de circuits et de chemins de données auto-contrôlables. Cet outil est implémenté en langage C et produit des descriptions structurelles et hiérarchiques en formats VHDL et Verilog. Dans les fichiers de sortie, les portes logiques sont représentées par des équations comportementales sans spécifications temporelles (équations logiques), et par conséquent ces fichiers sont indépendants d'une technologie quelconque. Les fichiers de sortie peuvent ensuite être synthétisés par une grande partie des environnements professionnels de CAO pour produire la même description d'origine mais correspondante à une technologie donnée ("technology mapping").

6.1. Structure générale de l'outil de CAO

La génération d'un circuit par l'outil s'effectue selon les étapes suivantes :

1. Lire les paramètres nécessaires pour la génération du circuit (type du circuit et ses dimensions).
2. Construire une chaîne d'éléments contenant les données nécessaires pour la description du circuit.
3. Ecrire la description du circuit en format HDL.

Les étapes 1 et 3 sont des interfaces d'utilisateur et n'ont pas besoin d'être développés ici. La deuxième étape consiste en la réalisation d'une structure de donnée simple représentant le circuit en utilisant le langage de programmation C. Cette structure comprend le nombre minimum d'informations nécessaires pour la description du circuit et sont : noms des cellules, nombre, type et dimensions des ports, nombre des connexions internes, la liste des composants et les connexions les reliant compris dans le circuit. Un circuit peut être une porte logique, une cellule de base (ex. : FA), un macro-bloc (ex. : un bloc générant les produits partiels d'un multiplieur), un opérateur (un multiplieur) ou un système (chemin de données). La figure 6.1 représente la structure générale de l'outil.

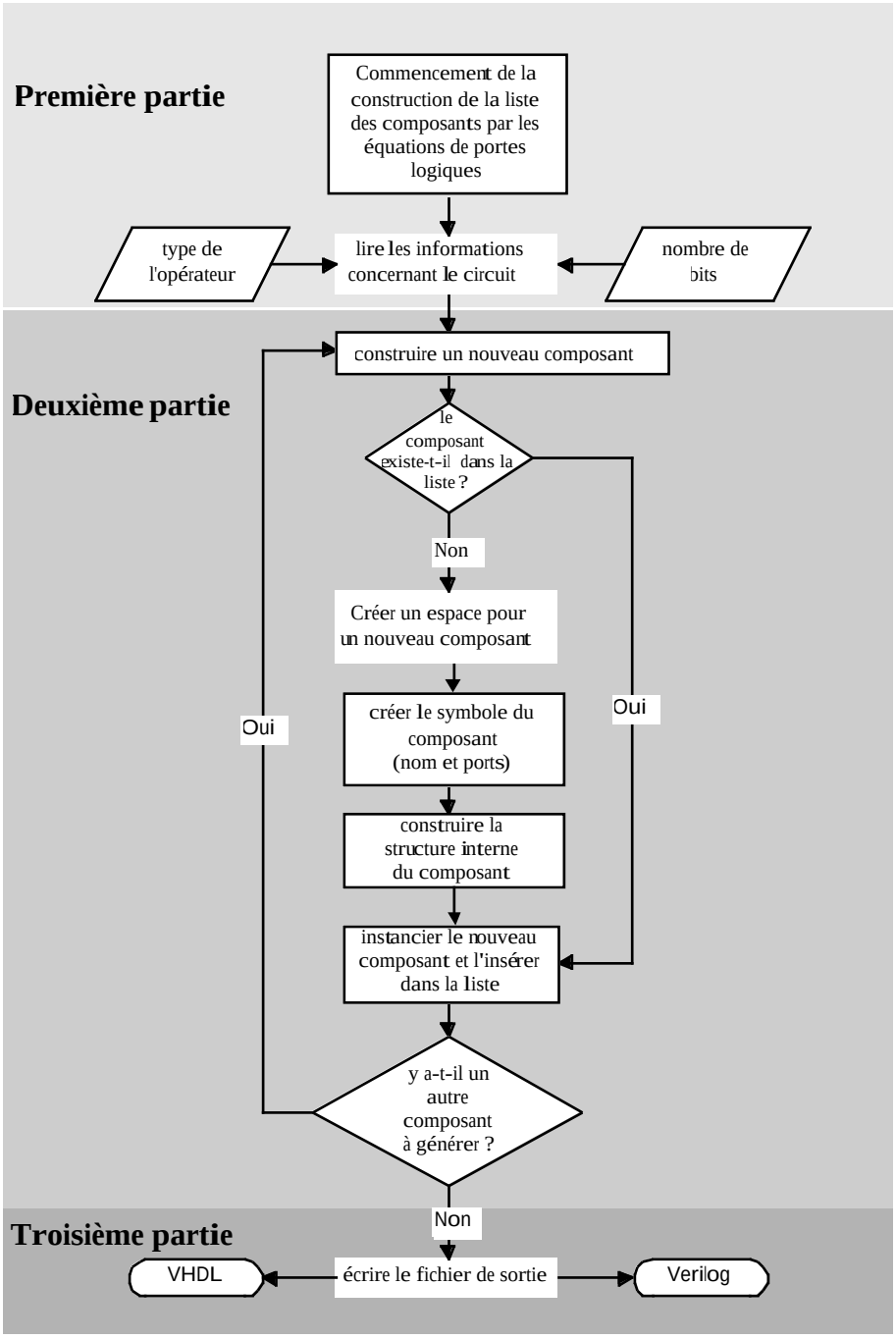


Figure 6.1. Structure générale de l'outil de CAO et flux des données.

6.2. Structure et Génération d'un Composant

6.2.1. Représentation d'un Composant

Tous les composants générés par l'outil ont la même structure de base. Cette structure est composée par les éléments (ou champs) suivants :

ID : Un nombre identifiant le composant. Les composants sont tous référencés par leurs numéros identifiants. Ces numéros sont divisés en trois groupes :

- de 0 à 1000 : les équations logiques.
- de 1001 à 10000 : les portes logiques et les cellules de base.
- de 10001 à 100000 : les opérateurs et systèmes.

Les portes logiques et les cellules de base sont déclarées sous la forme de noms correspondant à des numéros identifiants, alors que l'identification des opérateurs et systèmes est réalisée pendant l'exécution du programme.

Library Name : Le nom du composant utilisé pour écrire les fichiers VHDL/Verilog.

Ports : Une liste décrivant les ports externes du composant. Cette liste contient le nom, les dimensions et le type de chaque port.

Number of ports : Le nombre des ports externes du composant.

Net_list : Une liste décrivant les connexions internes du composant (numéro id, nom et dimensions).

Number of nets : Le nombre des connexions internes du composant.

Component list : Une liste des sous composants et les connexions les reliant du composant en construction.

Size : Les dimensions du composant en μm^2. Ce champ est calculé suivant le type et le nombre des sous composants du vecteur *Component lis*.

Vhdl_equation : L'équation en format VHDL du composant lorsque c'est une porte logique.

Verilog_equation : L'équation en format Verilog du composant lorsque c'est une porte logique.

6.2.2. Représentation des Connexions

Les connexions internes d'un composant sont représentées par les champs de la structure *net_list_element* suivante :

ID : Un nombre identifiant de la connexion.

Size : Les dimensions de la connexion.

Name : Le nom de la connexion utilisé pour écrire les fichiers VHDL/Verilog.

Une structure appelée *net_tuple* a aussi été déclarée pour manipuler les connexions d'un composant. Cette structure est composée par les champs suivants :

ID : Un nombre identifiant de la connexion.

Size : Les dimensions de la connexion.

6.3. Fonctions de Manipulation et de Génération des Composants

L'exécution du programme fait appel à trois différents groupes de fonctions.

6.3.1. Fonction de Lecture des Equations représentant les Portes logiques

Cette fonction lit l'ensemble des équations représentant les portes logiques utilisées et commence la construction d'une liste de composants en insérant les composants (les équations) lues l'une après l'autre dans l'ordre de leurs numéros identifiants.

6.3.2. Fonctions de Réalisation d'un Composant

Pour générer un composant, l'outil fait appel aux fonctions suivantes :

lookup_component (unsigned id) : Cette fonction vérifie si le composant dont le numéro d'identification est *id* existe dans la liste de composants en construction. Elle retourne la valeur *id* si le composant existe et zéro sinon.

*insert_component_in_lib(component_lib_type *comp)* :
Cette fonction permet d'insérer un composant *comp* dans la liste des composants formée pendant l'exécution du programme.

*clean_up_component(component_lib_type *comp)* :
Lors de la génération d'un composant, les dimensions des connexions et le nombre des ports externes ne sont connus que lorsque celui-ci est complètement généré. Pour cette raison, des dimensions assez larges sont allouées aux vecteurs des connexions et des ports externes. La fonction *clean_up_component* permet ensuite de réduire la taille de ces vecteurs aux tailles exactes.

*build_external_port(component_lib_type *base, char *name, unsigned direction, unsigned size)* :
Cette fonction crée des ports externes, leur attribue les noms, directions, et dimensions correspondants. Les ports externes sont tous réalisés avant les connexions internes.

*allocate_net(component_lib_type *base, unsigned size)* :
Cette fonction crée une connexion interne de dimension *size* et rend un pointeur de type *net_tuple* pour identifier et référencer cette connexion.

*instans_of_component(component_lib_type *base, unsigned id, net_tuple *connect)* :

Cette fonction instancie un composant dont l'identification est *id* et le relie avec une liste d'interconnexions.

*get_port(component_lib_type *comp, unsigned port_number)* :
Cette fonction rend le numéro d'identification du port numéro *port_number* du composant *comp*.

6.3.3. Fonctions d'Ecriture des Fichiers de Sortie

*write_files(char *name)* :
Cette fonction gère l'ouverture et la fermeture des fichiers de sortie et fait appel aux fonctions *write_VHDL* et *write_Verilog*.

*write_VHDL(char *filename)* :
Cette fonction transforme une représentation interne (la liste de composants construite) en une représentation VHDL structurelle.

*write_Verilog(char *filename)* :
Cette fonction transforme une représentation interne (la liste de composants construite) en une représentation Verilog structurelle.

6.4. Implémentation d'un Composant

6.4.1. Implémentation des Portes Logiques et des Cellules de Base

On a vu qu'une porte logique est représentée par son équation logique. Une porte logique est implémentée sous forme de fonction qui fait une instance de cette équation logique. En d'autres termes, une porte logique est un composant comprenant une liste d'un seul élément représentant l'équation logique de la porte. L'exemple suivant montre l'implémentation de la porte logique AND à deux entrées.

```
Component_lib_type *and2_cell(char *name)
{
component_lib_type *base;
net_tuple io_list[MAX_NETS];

base=new_component(name);
base->id=AND2_CELL;
```

```
build_external_port(base,"A",IN,1);
build_external_port(base,"B",IN,1);
build_external_port(base,"O",OUT,1);

instans_of_component(base, AND2_EQUATION, io_list);
return base;
}
```

Le code de programmation d'une porte AND

Cette porte a deux entrées appelées A et B et une sortie appelée O qui sont crées en utilisant la fonction *build_external_port*. Ces ports ont tous des dimensions égales à 1. L'équation logique représente ensuite la relation entre les entrées et la sortie.

Une cellule de base est à son tour composée de portes logiques et est implémentée sous forme de fonction en C. Cette fonction comporte les portes logiques qui la composent et la façon dont elles sont connectées. Ceci est réalisé en faisant appel aux fonctions de réalisation d'un composant présentées dans le paragraphe 3.2. La fonction de génération d'un FA est présentée par la suite comme un exemple sur la génération d'une cellule de base.

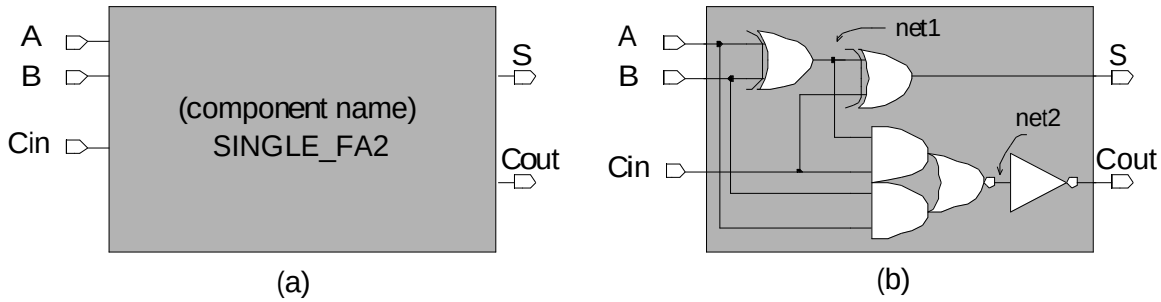


Figure 2. Implémentation d'une cellule de FA

```
component_lib_type *full_adder_type2(char *name)
{
/**** declaration of variables ****/
  component_lib_type *base;
  net_tuple io_list[MAX_NETS];

/**** Création du bloc symbolique de la cellule de base ****/
  base=new_component(name);
  base->id=SINGLE_FA2;

/**** Création des ports extérieurs ****/

  build_external_port(base,"A",IN,1);
  build_external_port(base,"B",IN,1);
  build_external_port(base,"CIN",IN,1);
  build_external_port(base,"S",OUT,1);
  build_external_port(base,"COUT",OUT,1);
```

```
/**** Réalisation de la structure interne ****/
  io_list[0]=get_port(base,1);
  io_list[1]=get_port(base,2);
  net1=allocate_net(base,1);
  io_list[2]=net1;
  instans_of_component(base,XOR2_CELL,io_list);

  io_list[0]=get_port(base,3);
  io_list[1]=net1;
  io_list[2]=get_port(base,4);
  instans_of_component(base,XOR2_CELL,io_list);

  io_list[0]=get_port(base,1);
  io_list[1]=get_port(base,2);
  io_list[2]=get_port(base,3);
  io_list[3]=net1;
  io_list[4]=net2;
  instans_of_component(base,AND2_AND2_NOR2_CELL,io_list);

  io_list[0]=net2;
  io_list[1]=get_port(base,5);
  instans_of_component(base,INVERTER,io_list);

  return base;
}
```

Le code de programmation de la cellule de FA

Tout d'abord, un nom est donné à la cellule et les ports externes sont ensuite créés. Un numéro d'identification interne (SINGLE_FA2) déclaré dans le fichier *gen_defines.h* est associé à la cellule et les ports (A,B,CIN,S,COUT) sont associés automatiquement avec les nombres identifiants internes à la cellule (1,2,3,4,5) respectivement. La variable *base* est un pointeur vers la liste complète des portes logiques utilisées dans la cellule.

La structure interne de la cellule est ensuite réalisée. Le vecteur de nœuds *io_list* représente la liste des ports externes des portes logiques à instancier. Chaque élément de ce vecteur est associé (connecté) soit à un port externe de la cellule soit à un nœud interne. Pour construire une liste de connexions correspondant à une porte logique quelconque correctement, on doit respecter l'ordre de déclaration des ports de cette porte. La première instanciation dans l'exemple précédent est une porte XOR. Cette porte est la porte de droite de la figure 6.2 et qui réalise la fonction logique (net1=A⊕B). Noter que le vecteur *io_list* est initié jusqu'à l'indice 2, ce qui fait un nombre total de nœuds égal à 3 comme exigé pour une porte XOR. La ligne *io_list[0]=get_port(base,1);* veut dire que le numéro d'identification du port 1 (port externe) du composant pointé par *base* (le FA en train de construction) est attribué au premier élément de la liste de nœuds *io_list*. La sortie de cette porte EXOR est un nœud interne qu'on doit créer. Ceci est fait en utilisant la fonction *allocate_net* qui crée un nouveau nœud interne de dimension 1 pour le composant pointé par *base* et donne à ce nœud le nom *net1*.

La fonction *instans_of_component* instancie ensuite la porte EXOR et l'insère dans la liste de composants pointée par *base* avec la liste d'interconnexions donnée par *io_list*. La construction de la cellule de FA continue ainsi jusqu'à l'instanciation de la porte logique inverseur.
L'implémentation du cœur de l'outil de CAO est donnée en annexe.

6.4.2. Implémentation de Macro-Blocs, Opérateurs et Chemins de Données

Le code de programmation des macro-blocs, des opérateurs et des chemins de données suit la même structure que celle présentée dans l'exemple précédent. Deux exemples de réalisation d'un macro-bloc et d'un opérateur sont donnés en annexe pour montrer la construction d'un chemin de données à partir des portes logiques et des cellules de base en utilisant les fonctions du générateur. Pour le cas d'un macro-bloc, l'exemple donné est celui d'un générateur de résidus utilisant le principe de la demi période de la base. Le cas de l'opérateur est un additionneur sûr en présence de fautes avec plusieurs bits de parité.

6.5. La Librairie de Circuits Auto-Contrôlables

Les circuits auto-contrôlables réalisés forment une librairie et sont implémentés dans l'outil sous la forme de fonctions. Ces fonctions font appel aux fonctions de réalisation d'un composant détaillées auparavant. Une liste des opérateurs implémentés est donnée en annexe.

Conclusions

Dans cette thèse nous avons étudié des opérateurs arithmétiques et logiques auto-contrôlables basés sur les codes de résidu et le code de parité. Dans un premier temps, nous avons démontré que les codes de résidu permettent d'obtenir des multiplieurs auto-contrôlables avec un coût supplémentaire très faible notamment pour les multiplieurs de grandes tailles. Dans certains cas, une combinaison d'un code de résidu de coût minimum et du code de parité permet d'éviter l'utilisation d'un code de résidu de base élevée, tout en gardant la sûreté en présence de fautes.

Nous avons ensuite généralisé des solutions sûres en présence de fautes d'additionneurs, de multiplieurs et de registres à décalage basés sur le code de parité. Les versions obtenues ont plusieurs bits de parité donnant ainsi un taux de couverture d’erreurs plus élevé pour des fautes multiples de type collage et pour des fautes transitoires, au prix d'une faible augmentation du coût.

Pour les fautes transitoires, dans le cas des additionneurs et des multiplieurs, nous avons aussi éliminé la duplication des retenues. Les solutions obtenues sont plus efficaces et moins coûteuses que les solutions sûres en présence de fautes de type collage.

Les techniques proposées sont flexibles d’un point de vue du contexte d’utilisation. Pour le cas des applications critiques en sécurité, les solutions utilisant plusieurs bits de parité avec duplication des retenues sont les mieux adaptées. Ces techniques offrent une couverture élevée face aux fautes permanentes (tels que les collages logiques et les fautes de délai). D’autre part, pour des applications n’exigeant pas un niveau de sécurité élevé mais utilisant des circuits fabriquées en technologies nanométriques (très sensibles aux fautes transitoires) ou évoluant dans des environnements hostiles, les solutions utilisant plusieurs bits de parité mais sans duplication des retenues seront les mieux adaptées. Finalement, pour des applications ayant des contraintes de sécurité, mais utilisant des technologies robustes ou évoluant dans des environnements non hostiles (faible niveau de bruit électromagnétique et faible flux de particules), la solution utilisant un seul bit de parité avec duplication des retenues sera la mieux adaptée.

Ces solutions ont été intégrées dans un outil informatique développé au sein de notre équipe, permettant ainsi de faciliter la tâche des concepteurs non spécialisés. L'outil en question

permet de générer une grande variété d'opérateurs arithmétiques et logiques auto-contrôlables de manière à donner une grande flexibilité pour la réalisation de chemins de données auto-contrôlables selon le type d'opérateur et le contexte de l'application. Les descriptions générées par l'outil sont écrites en langages VHDL et Verilog et sont indépendantes d'une technologie quelconque. Ces descriptions peuvent ensuite être utilisées par les outils de CAO professionnels. L'outil est mieux adapté pour la réalisation de générateurs de circuits à structures régulières et optimisées tels que les opérateurs arithmétiques et logiques étudiés dans cette thèse. Cette automatisation est une étape nécessaire pour faire admettre ces techniques par une majorité des concepteurs. L'étape suivante, consistant à l'industrialisation de ces outils semble être en bonne voie.

Bibliographie

[Ang00] L. Anghel, M. Nicolaidis et I. A. Noufal, "Self-Checking Circuits versus Realistic Faults in Very Deep Submicron", *Proceedings of VTS'00*, pp. 263-269, Montreal, May, 2000.

[And71] D. A. Anderson, "Design of self-checking digital networks using coding techniques", *Technical Report R-527*, CSL, Univ. of Illinois, IL, 1971.

[And73] D. A. Anderson et G. Metze, "Design of totally self-checking circuits for m-out-of-n codes", *IEEE Trans. on Computers*, vol. C-22, pp. 263-269, Mar. 1973.

[Ara89] M. Annaratone, *Digital CMOS Circuit Design*, Kluwer Academic Publishers, 1989.

[Boe97] E. Boehl, Th. Lindenkreuz et R. Stephan, "The Fail Stop Controller AE11", *ITC*, pp. 567-577, Nov. 1997.

[Boo51] A. D. Booth, "A signed binary multiplication technique", *Quarterly Journal of Mechanics and Applied Mathematics*, n° 4, pp. 236-240, June 1951.

[Bra63] E. L. Braun, *Digital Computer Design*, New-York Academic, 1963.

[Bre82] R. P. Brent et H. T. Kung, "A regular layout for parallel adders", *IEEE Transactions on Computers*, vol. C-31, n° 3, pp. 260-264, 1982.

[Bur94] B. Burgess, M. Alexander, Y.-W. Ho, S. P. Litch, et cols., "The PowerPC™ 603 microprocessor: A high performance, low power, superscalar RISC microprocessor", *Design Automation Conference - DAC'94*, pp. 300-306, 1994.

[Dad65] L. Dadda, "Some schemes for parallel multipliers", Alta Frequenza, vol. 19, pp. 349-356, March 1965.

[Dav78] R. David and P. Thevenod-Fosse, "Design of totally self-checking asynchronous modular circuits", *J. Des. Autom. Fault-Tolerant Computing*, vol. 2, pp. 271-278, October 1978.

[Dua97a] R. O. Duarte, H. Bederr, M. Nicolaidis et Y. Zorian, "Fault-secure shifter design: Results and implementations", in *European Design & Test Conference*, Paris, France, pp. 335-341, 17-20 March 1997.

[Dua97b] R. O. Duarte, H. Bederr, M. Nicolaidis et Y. Zorian, "Efficient totally self-checking shifter design", *JETTA - Journal of Electronic Testing - Special Issue*, Kluwer Academics Publishers, 1997.

[Dua97c] R. O. Duarte, " Techniques de Conception et Outils de CAO pour la Génération des Parties Opératives Auto-Contrôlables", Th. Microélectronique, TIMA, INPG, Grenoble 1997.

[Duf96] Jean-Louis Dufour, "Safety Computations in Integrated Circuits", *VTS*, , pp. 169-172, April 28-May 1, 1996.

[Efs94] C. Efstathiou, D. Nicolos, et J. Kalamatianos, "Area-Time Efficient Modulo 2^n-1 Adder Design", *IEEE Transactions on Circuits and Systems-II : Analog and Digital Signal Processing*, Vol. 41, No.7 pp. 169-172, July 1994.

[Gar68] O. N. Garcia et T. R. N. Rao, "On the method of checking logical operations", *Proceedings of 2nd Annual Princeton Conf. Inform. Sci. Sys.*, pp. 89-95, 1968.

[Han87] T. Han et D. A. Carlson, "Fast area-efficient VLSI adders", *8th Symposium on Computer*

Arithmetic, pp. 49-56, May 1987.

[Hen95] John L. Hennessy et David A. Patterson, *Architecture des Ordinateurs - Une approche quantitative*. Ediscience International, 1995.

[Hwa79] K. Hwang, *Computer Arithmetic, Principles, Architecture and Design*, John Wiley and Sons, New York - 1979.

[Kog73] P. M. Kogge and H. S. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations", *IEEE Transactions on Computers*, vol. C-22, n° 8, pp. 786-792, August 1973.

[Mak96] H. Makino, Y. Nakase, H. Suzuki, H. Morinaka, H. Shinohara et K. Mashiko, "An 8.8-ns 54 x 54-bit multiplier with high speed redundant binary architecture", *IEEE Journal of Solid-State Circuits*, vol. 31, n° 6, June 1996.

[McS61] O. L. McSorley, "High speed arithmetic in binary computers", in *Proceedings of the IRE*, pp. 67-91, January 1961.

[Mor91] J. Mori, M. Nagamatsu, M. Hirano, S. Tanaka, M. Noda, et al., "A 10-ns 54x54 parallel structured full array multiplier with 0.5 µm CMOS technology", *IEEE Journal of Solid-State Circuits*, vol. SC-26, n° 4, pp. 600-605, April 1991.

[Mul89] J.-M. Muller, *Arithmétique des Ordinateurs - Opérateurs et Fonctions Élémentaires*, Masson, Paris, 1989.

[Nic87] M. Nicolaidis, "Shorts in Self-Checking Circuits", International Test Conference, Washington DC, 1987.

[Nic88] M. Nicolaidis et B. Courtois, "Strongly code-disjoint checkers", *IEEE Trans. on Computers*, vol. C-37, pp. 751-756, June 1988.

[Nic93] M. Nicolaidis, "Efficient implementation of self-checking adders and ALUs", *Proc. 23th Fault Tolerant Computing Symposium*, Toulouse France, pp. 586-595 , June 1993

[Nic94] M. Nicolaidis, "Fault-secure property versus strongly code disjoint checkers", *IEEE Trans. on Comp.-Aided Design*, vol. 13, pp. 651-658, May 1994.

[Nic95] M. Nicolaidis, S. Manich et J. Figueras, "Achieving fault secureness in parity prediction array arithmetic operators", *1996 - European Design and Test Conference ED&TC*, Paris, March 1995.

[Nic97a] M. Nicolaidis, R.O. Duarte, S. Manich et J. Figueras, "Achieving fault secureness in parity prediction array arithmetic operators", *in IEEE Design & Test of Computers*, June 1997.

[Nic97b] M. Nicolaidis, "Carry Checking Parity Prediction Adders and ALUs", *IEEE Trans. on VLSI Systems*, 1997.

[Nic98a] M. Nicolaidis, "Scaling Deeper to Submicron: On-Line Testing to the Rescue", *In proceedings FTCS-28*, pp. 299-301, June 1998, Munich.

[Nic98b] M. Nicolaidis, "Design for Soft-Error Robustness to Rescue Deep Submicron Scaling", *In proceedings ITC 98*, October 1998, Washington DC.

[Nic98c] M. Nicolaidis, "On-Line Testing for VLSI: State of the Art and Trends", *Integration: the VLSI Journal*, Elsevier, Special Issue on "VLSI Testing toward 21 Century", Autumn 1998.

[Pas88] A.M. Paschalis, D. Nikolos, et C. Halatsis "Efficient Modular Design of TSC Checkers for m-

out-of-n Codes", *Integration: the VLSI Journal*, IEEE Trans. On Computer, vol. C-37, pp. 301-309, March 1988.

[Ped95] H. M. Pedersen, *Fault Tolerant Cell Library - Self-checking Adders/ALUs*, TIMA - Internal Report, July 1995.

[Pie94] S. J. Piestrak, "Design of Residue Generators and Multioperand Modular Adders Using Carry-Save Adders", IEEE Trans. On Computer, vol. 423, No. 1, pp. 68-77, Jan. 1994.

[Pie95] S. J. Piestrak, "Design of self-testing checkers for unidirectional error detecting codes", *Monografie Nr 24*, Oficyna Wydawnicza Politechniki Wroclawskiej, Wroclaw, 1995.

[Rao89] T. R. N. Rao and E. Fujiwara, *Error Control Coding for Computer Systems.* Prentice-Hall, Englewood Cliffs, N.J., 1989.

[Sat91] T. Sato, N. Nakajima, T. Sukemura et G. Goto, "A regularly structured 54-bit modified Wallace-tree multiplier", *VLSI Design Conference*, pp. 111-119, 1991

[Sel68] F. F. Sellers, M.-Y. Hsiao et L.W. Bearnson, *Error Detecting Logic for Digital Computers*, McGraw-Hill, New York, 1968.

[She77] J. J. Shedletsky, "Comment on the sequential and intermediate behavior of an end-around-carry adder", *IRE Trans. Comp.*, vol. C-26, pp. 271-272, Mar 1977.

[Skl60] J. Sklansky, "Conditional-sum addition logic", *IRE Transactions on Electronic Computers*, vol. EC-9, n° 2, pp. 226-231, June 1960.

[Smi77] J. E. Smith, "The design of totally self-checking check circuits for a class of unordered codes", *Journal of Design Autom. Fault-Tolerant Comput.*, vol.2, pp. 321-342, Oct, 1977.

[Smi78] J. E. Smith et G. Metze, "Strongly fault-secure logic networks", *IEEE Trans. on Computers*, vol. C-27, pp. 491-499, June 1978.

[Spa93] U. Sparmann, "On the Check Base Selection Problem for Fast Adders", *Proc. 11th VLSI Test Symp*. Atlantic City, NJ, April 1993.

[Spa96] U. Sparmann et S. M. Reddy, "On the effectiveness of residue code checking for parallel two's complement multipliers", *IEEE Transactions on VLSI Systems*, June 1996.

[Twa95] H. Al-Twaijry et M. Flynn, "Performance/Area tradeoffs in Booth multipliers", *Technical Report: CSL-TR-95-684*, November 1995.

[Wal64] C. S. Wallace, "A suggestion for a fast multiplier", *IEEE Transactions on Electronic Computers*, pp. 14-17, February 1964.

[Wak78] J. F. Wakerly, *Error Detecting Codes, Self-Checking Circuits and Applications*, North-Holland, New York, 1978.

[Wes94] N.H.E. Weste et K. Eshraghian, *Principles of CMOS VLSI Design - A Systems Perspective - second edition*, Addison-Wesley Publishing Co., 1994.

[Yu95] R. K. Yu et G. B. Zyner, "167 MHz radix-4 floating point multiplier", in *Proceedings 14th Symposium on Computer Arithmetic*, pp. 149-154, 1995.

Annexes

FS : fault secure
CR : basé sur le code résidu

La signification des éléments se trouve soit dans leurs appellations soit dans les figures présentées dans ce rapport.

A.1 - Liste des cellules de base
xor2_cell
xnor2_cell
or2_cell
or3_cell
nor2_cell
and2_and2_nor2_cell
or2_nand2_cell
and2_nor2_cell
and2_cell
and3_cell
and4_cell
nand2_cell
nand3_cell
nand4_cell
mux21_cell
mux41_cell

A.2 - Liste des macro-cellules
single_fadder1
single_fadder2
single_hadder
fa1_carry_dup
fa2_carry_dup
single_dual_rail_checker
block_b1
block_c1
block_c2
booth2_dec_cell
booth_dec_pp_cell
single_hadder_ci
ha_carry_dup
single_fadder2_ci
sum_fa
carry_fa2
carry_I_fa2
carry_fa1
carry_la1
carry_la2
carry_la3
carry_la4
carry_lapg
cellule_brent_kung1
cellule_brent_kung2
cellule_brent_kung3

A. 3 - Liste des macro-blocs
n_dual_rail_checker
translator
mod_A_period
mod_A_period_buffers
mod_A_Hperiod
multi_operand_modular_adder
MOMM_2_A
modulo_A_adder
parity_tree
or_slice
and_slice
and_slice_n_inputs
or_tree
and_tree
and_tree_n_inputs
n_carry_look_ahead
MOMM_2_A
A_x_B_modulo_Hperiode
macro_block_b1
macro_block_b1_for_wallace
booth2_decoder
slice_hadder
slice_fadder1
slice_rca
slice_generate_propagate

A. 4 - Liste des opérateurs
mult_braun
mult_wallace
mult_wallace_parity_predict
multiplier_fs_cr
booth2_braun
booth2_braun_double_recoder
bo2_wa_double_rec
n_kogge_stone
n_Brent_Kung
n_Han_Carlson
n_sklanski
final_adder
adder_fs_cr
booth2_wallace
wallace_fs_cr
multiplier_booth_braun_fs_cr
adder_fs_with_groups_of_parity
pc_booth2_wallace
adder_fs_with_groups_of_parity
bo_wa_fs_cr
sc_shifter
shifter

gen_define.h

```
#define FALSE 0
#define TRUE 1
#define EVEN 0
#define ODD 1

#define FIRST 0
#define SECOND 1
#define THIRD 2
#define FOURTH 3
#define LAST 4
#define OTHERS 5

#define NEW_NET 0
#define OPEN_PORT 65535
#define MAX_NETS 10000
#define MAX_PORTS 100
#define MAX_DIFFERENT_COMPONENTS
500
#define MAX_COMPONENTS 10000

#define IN 1
#define OUT 2
#define INOUT 3

#define ALU_START 5
#define ALU1_START 6

#define XOR2_EQUATION 1
#define CARRY1_EQUATION 2
#define CARRY2_EQUATION 3
#define SUM_EQUATION 4
#define AND2_EQUATION 5
#define INVERTER_EQUATION 6
#define XNOR2_EQUATION 7
#define OR2_EQUATION 8
#define NAND2_EQUATION 9
#define NOR2_EQUATION 10
#define AND3_EQUATION 11
#define AND4_EQUATION 12
#define NAND3_EQUATION 13
#define NAND4_EQUATION 14
#define NOR3_EQUATION 15
#define AND2_AND2_NOR2_EQUATION 16
#define CARRY2_I_EQUATION 17
#define BUFER_EQUATION 18
#define OR2_NAND2_EQUATION 19
#define AND2_NOR2_EQUATION 20
#define ROM_EQUATION 21
#define OR3_EQUATION 22
#define MUX_EQUATION 23
#define MUX21_EQUATION 24
#define MUX41_EQUATION 25
#define HAMMING_MUX_EQUATION 26


#define AND2_CELL 1001
#define XOR2_CELL 1002
#define MUX21_CELL 1003
#define MUX41_CELL 1004
#define INVERTER 1005
#define XNOR2_CELL 1006
#define OR2_CELL 1007
#define NAND2_CELL 1008
#define NOR2_CELL 1009
#define AND2_AND2_NOR2_CELL 1010
#define AND3_CELL 1011
#define AND4_CELL 1012
#define NAND3_CELL 1013
#define NAND4_CELL 1014
#define NOR3_CELL 1015
#define BOOTH_DEC_CELL 1018
#define BOOTH_DEC_PP_CELL 1020
#define BLOCK_C1 1021
#define BLOCK_C2 1022
#define BUFER 1024
#define OR2_NAND2_CELL 1025
#define AND2_NOR2_CELL 1026
#define OR3_CELL 1027


#define CARRY_DUP_FA 1301
#define CARRY_FA1 1302
#define CARRY_FA2 1303
#define SUM_FA 1304
#define CARRY_I_FA2 1305

#define ROM 2000


#define SINGLE_FADDER1 5001
#define SINGLE_FADDER2 5002
#define SINGLE_HADDER 5003
#define CARRY_LA1 5005
#define CARRY_LA2 5006
#define CARRY_LA3 5007
#define CARRY_LA4 5008
#define CARRY_LAPG 5009
#define MULTIPLIER_2_2 5010
#define SINGLE_DUAL_RAIL_CHECKER
5011
#define HA_CARRY_DUP 5012
#define FA1_CARRY_DUP 5013
#define FA2_CARRY_DUP 5014

#define SINGLE_FADDER2_CI 5016
#define SINGLE_HADDER_CI 5017

#define CELLULE_BRENT_KUNG1 5018
#define CELLULE_BRENT_KUNG2 5019
#define CELLULE_BRENT_KUNG3 5020
#define BLOCK_B1 5021
#define BOOTH2_DEC_CELL 5022

#define SLICE_HADDER 5023
#define SLICE_FADDER1 5024
#define SLICE_RCA 5025
#define SLICE_HADDER1 5026
#define SLICE_FADDER11 5027
#define SLICE_RCA1 5028
```

```
#define ONE_OPERATION_SHIFTER 1
#define TWO_OPERATIONS_SHIFTER 2
#define FOUR_OPERATIONS_SHIFTER 3

#define ROTATION 1
#define LOGIC_LEFT_SHIFT 2
#define LOGIC_RIGHT_SHIFT 3
#define ARITHMETIC_SHIFT 4

#define VHDL_SIGNAL "std_logic"
#define VHDL_LIBRARY "IEEE"
#define VHDL_USE
"IEEE.STD_LOGIC_1164.ALL"

typedef struct {
  unsigned number;
  unsigned size;
} net_tuple;

typedef struct
{
  unsigned number;
  unsigned size;
  char *name;
} net_list_element;

typedef struct component_type
{
  unsigned component_id;
  unsigned *port_array;
  struct component_type *next;
};

typedef struct
{
  char *name;
  unsigned type;
  unsigned size;
} component_port_type;

typedef struct
{
  unsigned id;
  char *library_name;
  char *vhdl_equation;
  char *verilog_equation;
  unsigned num_ports;
  double size;
  component_port_type *ports;
  net_list_element *nets_list;
  unsigned num_nets;
  struct component_type *comp_list;
} component_lib_type;

typedef struct
{
  unsigned id;
  component_lib_type *comp_pointer;
} comp_lib_list_type;

typedef struct {
  unsigned list[4];
} prog;
```

generator.c (le cœur de l'outil)

```
#include"gen_defines.h"
#include"multi_file.h"
#include<stdio.h>
#include<string.h>
#include<math.h>


  /* The library of available components */
comp_lib_list_type
component_lib[MAX_COMPONENTS];
  /* Number of components in data-base */
unsigned num_comp=0;

unsigned lookup_component(unsigned id)
{
  unsigned n;

  for (n=0;((n<num_comp) &&
(component_lib[n].id!=id));n++);
  if (component_lib[n].id!=id)
    return 0;
  else
    return n+1;
}

void clean_up_component(component_lib_type
*base)
{
  char temp_string[255];
  unsigned n;
  component_port_type *temp_port;
  net_list_element *temp_net;
  struct component_type *temp_comp;

  /* Reduce nets list, to fit actual size */
  temp_net=new net_list_element[base->num_nets];
  for (n=0;n<base->num_nets;n++)
    temp_net[n]=base->nets_list[n];
  delete base->nets_list;
  base->nets_list=temp_net;

  /* Reduce external port array to fit actual size */
  temp_port=new component_port_type[base-
>num_ports];
  for (n=0;n<base->num_ports;n++)
    temp_port[n]=base->ports[n];
  delete base->ports;
  base->ports=temp_port;

  /* Assign names to internal nets */
  for (n=base->num_ports;n<base->num_nets;n++)
  {
    sprintf(temp_string,"NET%d",n);
    base->nets_list[n].name=new
char[strlen(temp_string)+1];
    strcpy(base->nets_list[n].name,temp_string);
  }
  /* Calculate overall size */
  temp_comp=base->comp_list;
  base->size=0.0;
  while (temp_comp!=NULL)
  {
    n=lookup_component(temp_comp-
>component_id);
    base->size+=(component_lib[n-1].comp_pointer)-
>size;
    temp_comp=temp_comp->next;
  }

}

void insert_component_in_lib(component_lib_type
*comp)
{
  unsigned n, m,k=1;

  if (comp->nets_list!=NULL)
    clean_up_component(comp);
  if (num_comp==0)
  {
    component_lib[0].id=comp->id;
    component_lib[0].comp_pointer=comp;
    num_comp=1;
  }else{
    for(n=0;component_lib[n].id<10001;n++);
    for(;n<num_comp;n++) {
      if(!strcmp(component_lib[n].comp_pointer-
>library_name,comp->library_name)) {
/* printf("\n comp->library_name %s= \n",comp-
>library_name); */
/* printf("\n yes\n"); */
/* printf("\n component_lib[n].id=%d\n",comp->id);
*/
            comp->id=component_lib[n].id;
            k=0;
            break;
      }
    }
    if(k==1) {
      n=0;
      while
((n<num_comp)&&(component_lib[n].id<comp-
>id))
            n++;
      if (n!=num_comp)
            {
              for (m=num_comp;m>n;m--)
        component_lib[m]=component_lib[m-1];
            }
      component_lib[n].id=comp->id;
      component_lib[n].comp_pointer=comp;
      num_comp++;
    }
  }
}

void insert_equation_component_in_lib()
{
```

```
 unsigned n, end_of_components=FALSE;
 component_lib_type *temp_comp;

 for (n=1;n<1000;n++)
 {
  switch(n)
  {
   case INVERTER_EQUATION :
     temp_comp=new component_lib_type;
     temp_comp->id=INVERTER_EQUATION;
     temp_comp->library_name=NULL;
     temp_comp->num_ports=0;
     temp_comp->size=1.0;
     temp_comp->ports=NULL;
     temp_comp->nets_list=NULL;
     temp_comp->num_nets=0;
     temp_comp->comp_list=NULL;
     temp_comp->vhdl_equation = new
char[strlen("O <= (not A)")+1];
     temp_comp->verilog_equation = new
char[strlen("O = (~A)")+1];
     strcpy(temp_comp->vhdl_equation,"O <= (not
A)");
     strcpy(temp_comp->verilog_equation,"O =
(~A)");
     break;
    case ROM_EQUATION :
     temp_comp=new component_lib_type;
     temp_comp->id=ROM_EQUATION;
     temp_comp->library_name="rom";
     temp_comp->num_ports=0;
     temp_comp->size=1.0;
     temp_comp->ports=NULL;
     temp_comp->nets_list=NULL;
     temp_comp->num_nets=0;
     temp_comp->comp_list=NULL;
     temp_comp->vhdl_equation = new
char[strlen("data <=
bitv_to_stdv(rom_t(slv_to_int(addr)))")+1];
     temp_comp->verilog_equation = new
char[strlen("data =
bitv_to_stdv[rom_t[slv_to_int[addr]]]")+1];
     strcpy(temp_comp->vhdl_equation,"data <=
bitv_to_stdv(rom_t(slv_to_int(addr)))");
     strcpy(temp_comp->verilog_equation,"data =
bitv_to_stdv[rom_t[slv_to_int[addr]]]");
     break;
    case HAMMING_MUX_EQUATION :
     temp_comp=new component_lib_type;
     temp_comp-
>id=HAMMING_MUX_EQUATION;
     temp_comp->library_name="hamming_mux";
     temp_comp->num_ports=0;
     temp_comp->size=1.0;
     temp_comp->ports=NULL;
     temp_comp->nets_list=NULL;
     temp_comp->num_nets=0;
     temp_comp->comp_list=NULL;
     temp_comp->vhdl_equation = new
char[strlen("correction_bits <=
rom_t(slv_to_int(error_control))")+1];
     temp_comp->verilog_equation = new
char[strlen("correction_bits =
rom_t[slv_to_int[error_control]]")+1];
     strcpy(temp_comp-
>vhdl_equation,"correction_bits <=
rom_t(slv_to_int(error_control))");
     strcpy(temp_comp-
>verilog_equation,"correction_bits =
rom_t[slv_to_int[error_control]]");
     break;
case NOR2_EQUATION :
     temp_comp=new component_lib_type;
     temp_comp->id=NOR2_EQUATION;
     temp_comp->library_name=NULL;
     temp_comp->num_ports=0;
     temp_comp->size=1.0;
     temp_comp->ports=NULL;
     temp_comp->nets_list=NULL;
     temp_comp->num_nets=0;
     temp_comp->comp_list=NULL;
     temp_comp->vhdl_equation = new
char[strlen("O <= not(A or B)")+1];
     temp_comp->verilog_equation = new
char[strlen("O = ~(A | B)")+1];
     strcpy(temp_comp->vhdl_equation,"O <= not(A
or B)");
     strcpy(temp_comp->verilog_equation,"O = ~(A
| B)");
     break;

.
.
.
.

default :
     end_of_components=TRUE;
  }
  if(!end_of_components)
   insert_component_in_lib(temp_comp);
  else
   break;
 }

}

/*****************************************
*************************************/
/* Functions for manipulating the internal data-
structure                */
/*****************************************
*************************************/

net_tuple build_external_port(component_lib_type
*base, char *name, unsigned dir, unsigned size)
/* Assigns net numbers to an external port, and saves
the name in the list of nets */
/* It is assumed that ALL external ports are allocated
before ANY internal nets */

{
```

```
  net_tuple temp;

  /* Allocate space for external ports */
  if (base->ports==NULL)
    base->ports=new
component_port_type[MAX_PORTS];

  /* Insert net number (and eventually allocate space)
*/
  if (base->num_nets==0)
  {
    if (base->nets_list==NULL)
      base->nets_list=new
net_list_element[MAX_NETS];
    base->nets_list[0].number=1;
  }
  else
    base->nets_list[base->num_nets].number=base-
>nets_list[(base->num_nets)-1].number+
                                base->nets_list[(base-
>num_nets)-1].size;

  /* Insert port name */
  (base->ports[base->num_ports]).name=new
char[strlen(name)+1];
  strcpy((base->ports[base-
>num_ports]).name,name);
  if (dir==OUT)
  {
    (base->nets_list[base->num_nets]).name=new
char[strlen(name)+6];
    strcpy((base->nets_list[base-
>num_nets]).name,name);
    strcat((base->nets_list[base-
>num_nets]).name,"_READ");
  }
  else
  {
    (base->nets_list[base->num_nets]).name=new
char[strlen(name)+1];
    strcpy((base->nets_list[base-
>num_nets]).name,name);
  }

  /* Insert port size */
  base->ports[base->num_ports].size=size;
  base->nets_list[base->num_nets].size=size;

  /* Insert port direction */
  base->ports[base->num_ports].type=dir;

  /* Copy information onto temp */
  temp.number=base->nets_list[base-
>num_nets].number;
  temp.size=size;

  /* Update sizes of lists */
  base->num_nets++;
  base->num_ports++;

  return temp;
}

net_tuple allocate_net(component_lib_type *base,
unsigned size)
{
  net_tuple temp;

  base->nets_list[base->num_nets].number=base-
>nets_list[(base->num_nets)-1].number+
                                base->nets_list[(base-
>num_nets)-1].size;
  base->nets_list[base->num_nets].size=size;
  base->nets_list[base->num_nets].name=NULL;

  temp.size=base->nets_list[base->num_nets].size;
  temp.number=base->nets_list[base-
>num_nets].number;
  base->num_nets++;
  return temp;
}

component_lib_type *generate_component(unsigned
id)
{
  component_lib_type *temp;

  if ((id<1000) || (id>10000))
  {
    printf("\nError - trying to generate component,
with component id : %d\n\n",id);
    /* exit(3); */
  }
  switch (id)
  {
    case CARRY_FA1 :
temp=carry_fa1("CARRY_FA1");
                break;
    case SINGLE_FADDER1 :
temp=single_fadder1("SINGLE_FADDER1");
                break;
    case SINGLE_FADDER2_CI :
temp=single_fadder2_ci("SINGLE_FADDER2_CI");
                break;
    case SINGLE_FADDER2 :
temp=single_fadder2("SINGLE_FADDER2");
                break;
    case SINGLE_HADDER :
temp=single_hadder("SINGLE_HADDER");
                break;
    case SINGLE_HADDER_CI :
temp=single_hadder_ci("SINGLE_HADDER_CI");
                break;
    case FA1_CARRY_DUP :
temp=fa1_carry_dup("FA1_CARRY_DUP");
                break;
    case FA2_CARRY_DUP :
temp=fa2_carry_dup("FA2_CARRY_DUP");
                break;
case CARRY_LAPG :
temp=carry_lapg("CARRY_LAPG");
                break;
```

```
   case SINGLE_DUAL_RAIL_CHECKER :
temp=single_dual_rail_checker("SINGLE_DUAL_R
AIL_CHECKER");
                break;
   case CARRY_FA2 :
temp=carry_fa2("CARRY_FA2");
                break;
   case CARRY_I_FA2 :
temp=carry_I_fa2("CARRY_I_FA2");
                break;
   case SUM_FA : temp=sum_fa("SUM_FA");
                break;
   case XOR2_CELL :
temp=xor2_cell("XOR2_CELL");
                break;
   case XNOR2_CELL :
temp=xnor2_cell("XNOR2_CELL");
                break;
   case OR2_CELL : temp=or2_cell("OR2_CELL");
                break;

.
.
.
.

default : printf("\nError - trying to generate
component, with component id : %d\n\n",id);

  }
  insert_component_in_lib(temp);
  return temp;
}

struct component_type
*instans_of_component(component_lib_type *base,
                                unsigned id, net_tuple
*connect)
{
  struct component_type *temp;
  component_lib_type *comp;
  unsigned n, m;

  /* Allocate space for new component, and insert in
beginnning of component list */
  temp = new (struct component_type);
  temp->next=base->comp_list;
  base->comp_list=temp;

  /* Find library component */
  n=lookup_component(id);
  if (n==0)
    comp=generate_component(id);
  else
    comp=component_lib[n-1].comp_pointer;

  /* Connect ports */
  temp->component_id=id;
  temp->port_array = new unsigned[(comp-
>num_ports)];
  for (m=0;m<(comp->num_ports);m++)
  {
   if (connect[m].number==NEW_NET)
   {
    base->nets_list[base->num_nets].number=base-
>nets_list[(base->num_nets)-1].number+
                                base->nets_list[(base-
>num_nets)-1].size;
    base->nets_list[base->num_nets].size=comp-
>ports[m].size;
    temp->port_array[m]=base->nets_list[base-
>num_nets].number;
    base->nets_list[base->num_nets].name=NULL;
    base->num_nets++;
   }
   else
   {
    temp->port_array[m]=connect[m].number;
    if (connect[m].number!=OPEN_PORT)
    {
     /* Check the net size and boundary */
     if (connect[m].size!=comp->ports[m].size)
     {
      printf("Unequal net sizes in requested
connection in instans_of_component");
      printf("\nComponent name : %s\n",comp-
>library_name);
      printf("Connection number : %d\n",m);
     /*  exit(3); */
     }
     n=0;
     while (((base->nets_list[n].number) <
(connect[m].number)) && (n<(base->num_nets-1)) )
          n++;
     if ((base->nets_list[n].number) >
(connect[m].number))
      n--; /* Counted too far in the nets list - it was
the previous signal */
     if ((base->nets_list[n].number+base-
>nets_list[n].size) <
(connect[m].number+connect[m].size))
          {
      printf("Connection across net boundary in
instans_of_component");
      printf("\nComponent number : %d\n",id);
    /*   exit(3); */
          }
    }
   }
  }

  return temp;
}

net_tuple get_net(struct component_type *comp,
unsigned port_number)
{
  net_tuple temp;
  unsigned n;
  component_lib_type *temp_comp;

  temp.number=comp->port_array[port_number-1];
```

```
  n=lookup_component(comp->component_id);
  temp_comp=component_lib[n-1].comp_pointer;
  temp.size=temp_comp->ports[port_number-1].size;
  return temp;
}

net_tuple get_port(component_lib_type *base,
unsigned external_port_number)
{
  net_tuple temp;

  temp.number=base-
>nets_list[external_port_number-1].number;
  temp.size=base->nets_list[external_port_number-
1].size;
  return temp;
}


/*****************************************
**********************************/


/*****************************************
**********************************/
/* Writes the files (Verilog and VHDL)
*/
/*****************************************
**********************************/

void write_VHDL(FILE *vhdl_file,
component_lib_type *comp, unsigned *diff_comp)
{
  unsigned n, p, q, id;
  struct component_type *temp;
  component_lib_type *temp_comp;
  id=comp->id;
  FILE *rom_d,*mux_d,*hamming_mux_d;
  char rom_string[10],temp_string[200];

  if(strstr(comp->library_name,"rom")) {
    sprintf(rom_string,"donnees%c",comp-
>library_name[4]);
    if((rom_d=fopen(rom_string,"r"))==NULL) {
      printf("error while opening file");
      exit(1);
    }
    while(!feof(rom_d)) {
      fgets(temp_string,150,rom_d);
      fputs(temp_string,vhdl_file);
    }
    fclose(rom_d);
  }

  if(strstr(comp->library_name,"mux")) {
    sprintf(rom_string,"mux%c_d%c",comp-
>library_name[3],comp->library_name[5]);
    if((mux_d=fopen(rom_string,"r"))==NULL) {
      printf("error while opening file");
      exit(1);
    }
    while(!feof(mux_d)) {
      fgets(temp_string,150,mux_d);
      fputs(temp_string,vhdl_file);
    }
    fclose(mux_d);
  }

  if(strstr(comp-
>library_name,"hamming_corrector")) {
    sprintf(rom_string,"hamming_donnees%c",comp-
>library_name[17]);
    if((rom_d=fopen(rom_string,"r"))==NULL) {
      printf("error while opening file");
      exit(1);
    }
    while(!feof(rom_d)) {
      fgets(temp_string,150,rom_d);
      fputs(temp_string,vhdl_file);
    }
    fclose(rom_d);
  }

  /* Writing of header and entity declaration */
  fputs("-- VHDL file generated by Generator II --
",vhdl_file);
  fprintf(vhdl_file,"\n-- Estimated area (std. cells
only) : %lf --",comp->size);
  fprintf(vhdl_file,"\nlibrary ");
  fprintf(vhdl_file,VHDL_LIBRARY);
  fprintf(vhdl_file,";\nuse ");
  fprintf(vhdl_file,VHDL_USE);
  fprintf(vhdl_file,";\n\nentity %s is",comp-
>library_name);
  fprintf(vhdl_file,"\n  port(");
  for (n=0;n<comp->num_ports;n++)
  {
    fprintf(vhdl_file,"%s : ",comp->ports[n].name);
    switch (comp->ports[n].type)
    {
      case IN : fprintf(vhdl_file,"in ");
                break;
      case OUT : fprintf(vhdl_file,"out ");
                break;
      case INOUT : fprintf(vhdl_file,"inout ");
                break;
    }
    fprintf(vhdl_file,VHDL_SIGNAL);
    if (comp->ports[n].size>1)
      fprintf(vhdl_file,"_vector(%d downto 0)",comp-
>ports[n].size-1);
    if (n<(comp->num_ports-1))
      fputs(";\n       ",vhdl_file);
  }

  fprintf(vhdl_file,");\nend %s;\n\n",comp-
>library_name);

  fprintf(vhdl_file,"architecture structure of %s
is\n\n",comp->library_name);

  if (id>2000) {
```

```
  /* Declare components used in design */
  for(n=0;diff_comp[n]!=0;n++)
  {
    temp_comp=component_lib[diff_comp[n]-
1].comp_pointer;
    if((!strstr(comp-
>library_name,"rom"))&&(!strstr(comp-
>library_name,"mux"))&&(!strstr(comp-
>library_name,"hamming_corrector"))) {
      fprintf(vhdl_file,"  component %s\n",temp_comp-
>library_name);
      fprintf(vhdl_file,"    port(");
      for (p=0;p<temp_comp->num_ports;p++)
            {
              fprintf(vhdl_file,"%s : ",temp_comp-
>ports[p].name);
              switch (temp_comp->ports[p].type)
                {
                case IN : fprintf(vhdl_file,"in ");
                break;
                case OUT : fprintf(vhdl_file,"out ");
                 break;
                case INOUT : fprintf(vhdl_file,"inout ");
                  break;
                }
              fprintf(vhdl_file,VHDL_SIGNAL);
              if ((temp_comp->ports[p]).size>1)
                fprintf(vhdl_file,"_vector(%d downto
0)",(temp_comp->ports[p]).size-1);
              if (p<(temp_comp->num_ports-1))
                fputs(";\n         ",vhdl_file);
            }
      fprintf(vhdl_file,");\n  end component;\n\n");
    }
  }

  /* Declare internal signals used in design */
  if((!strstr(comp-
>library_name,"rom"))&&(!strstr(comp-
>library_name,"mux"))&&(!strstr(comp-
>library_name,"hamming_corrector"))) {
    for (n=0;n<comp->num_nets;n++)
      {
            if ((n>=comp->num_ports) || (comp-
>ports[n].type==OUT))
              {
                /* And write as a signal in VHDL file */
                fprintf(vhdl_file,"  signal %s : ",comp-
>nets_list[n].name);
                fprintf(vhdl_file,VHDL_SIGNAL);
                if ((comp->nets_list[n].size) > 1)
                  fprintf(vhdl_file,"_vector(%d downto
0)",comp->nets_list[n].size-1);
                fprintf(vhdl_file,";\n");
              }
      }
  }

  } /****** end if id *********/
```

```
   /* Write the architecture */
  fprintf(vhdl_file,"\nbegin\n");

  temp=comp->comp_list;
  if((id>2000)&&(!strstr(comp-
>library_name,"rom"))&&(!strstr(comp-
>library_name,"mux"))&&(!strstr(comp-
>library_name,"hamming_corrector"))) {
   /* Connect _READ signals to ports */
   for (n=0;n<comp->num_ports;n++)
   {
    if (comp->ports[n].type==OUT)
    {
      fprintf(vhdl_file,"  %s <= %s;\n",comp-
>ports[n].name,comp->nets_list[n].name);
    }
   }

   n=1; /* Component instans number */
   while (temp!=NULL)
   {
    p=lookup_component(temp->component_id);
    temp_comp=component_lib[p-1].comp_pointer;
    fprintf(vhdl_file,"  I%d : %s\n",n,temp_comp-
>library_name);
    fprintf(vhdl_file,"    port map(");
    for (p=0;p<temp_comp->num_ports;p++)
    {
     /* Write current port name of component - ie.
named association */
      fprintf(vhdl_file,"%s => ",temp_comp-
>ports[p].name);

      if (temp->port_array[p]==OPEN_PORT)
       fprintf(vhdl_file,"OPEN");
      else
      {
       /* q = index of first net in nets_list, with number
>= the net we
          want to connect */

       q=0;
       while (((comp->nets_list[q].number) < (temp-
>port_array[p])) && (q<(comp->num_nets-1)) )
            q++;
       if ((comp->nets_list[q].number) > (temp-
>port_array[p]))
         q--; /* Counted too far in the nets list - it was
the previous signal */
       fprintf(vhdl_file,"%s",comp-
>nets_list[q].name);

       if ((temp_comp->ports[p].size)>1)
       {
        /* Connecting a bus */
        fprintf(vhdl_file,"(%d downto ",((temp-
>port_array[p])+
                                (temp_comp-
>ports[p].size)-
                                (comp-
>nets_list[q].number))-1 );
```

```
        fprintf(vhdl_file,"%d)",((temp->port_array[p])-
(comp->nets_list[q].number)));
       }
       else if ((comp->nets_list[q].size)!=1)
         fprintf(vhdl_file,"(%d)",((temp-
>port_array[p])-(comp->nets_list[q].number)) );
     }
     if (p<(temp_comp->num_ports-1))
      fprintf(vhdl_file,", ");
   }
   fprintf(vhdl_file,");\n");
   temp=temp->next;
   n++;
  }

  } /****** end if id *********/ else {

fprintf(vhdl_file,"%s",component_lib[diff_comp[0]-
1].comp_pointer->vhdl_equation);
   fprintf(vhdl_file,";\n");
  }
  fprintf(vhdl_file,"end structure;\n\n");
}

/*****************************************
*****************/

void write_verilog(FILE *verilog_file,
component_lib_type *comp)
{
  unsigned n, m, p, q, id;
  struct component_type *temp;
  component_lib_type *temp_comp;

  id=comp->id;
  /* Writing of header and module declaration */
  fputs("// Verilog file generated by Generator II
//",verilog_file);
  fprintf(verilog_file,"\n// Estimated area (std. cells
only) : %lf //",comp->size);
  fprintf(verilog_file,"\n\nmodule %s(",comp-
>library_name);
  for (n=0;n<comp->num_ports;n++)
  {
   fprintf(verilog_file,"%s",comp->ports[n].name);
   if (n<(comp->num_ports-1))
    fprintf(verilog_file,", ");
  }
  fprintf(verilog_file,");\n");

  /* Writing of port types */
  for (n=0;n<comp->num_ports;n++)
  {
   switch (comp->ports[n].type)
   {
    case IN : fprintf(verilog_file,"input ");
              break;
    case OUT : fprintf(verilog_file,"output ");
               break;
    case INOUT : fprintf(verilog_file,"inout ");
                 break;
   }
   if ((comp->nets_list[n]).size>1)
    fprintf(verilog_file,"[%d:0]",comp->ports[n].size-
1);
   fprintf(verilog_file,"%s;\n",comp->ports[n].name);
  }

  /* Declare internal signals used in design */
  if (id>2000) {

  fprintf(verilog_file,"\n");
  for (n=comp->num_ports;n<comp->num_nets;n++)
  {
   /* And write as a signal in verilog file */
   fprintf(verilog_file,"wire ");
   if (comp->nets_list[n].size>1)
    fprintf(verilog_file,"[%d : 0]",comp-
>nets_list[n].size-1);
   fprintf(verilog_file,"%s;\n",comp-
>nets_list[n].name);
  }
  }

  /* Writing of library components and connections */
  fprintf(verilog_file,"\n");
  temp=comp->comp_list;

  if (id>2000) {

  n=1;
  while (temp!=NULL)
  {
   m=lookup_component(temp->component_id);
   temp_comp=component_lib[m-1].comp_pointer;
   fprintf(verilog_file,"%s I%d(",temp_comp-
>library_name,n);
   for (p=0;p<temp_comp->num_ports;p++)
   {
    fprintf(verilog_file,".%s(",temp_comp-
>ports[p].name);

    if (temp->port_array[p]==OPEN_PORT)
     fprintf(verilog_file,")");
    else
    {
     q=0;
     while (((comp->nets_list[q].number)<(temp-
>port_array[p])) && (q<(comp->num_nets-1)))
          q++;
      /* If counted too far, subtract one from q - ig.
the previous signal */
     if ((comp->nets_list[q].number)>(temp-
>port_array[p])) q--;
     if ((q<comp->num_ports) && (comp-
>ports[q].type==OUT))
       fprintf(verilog_file,"%s",comp-
>ports[q].name);
     else
       fprintf(verilog_file,"%s",comp-
>nets_list[q].name);
```

```
      if ((temp_comp->ports[p].size)>1)
      {
       /* Connecting a bus */
       fprintf(verilog_file,"[%d : ",((temp-
>port_array[p])+
                          (temp_comp->ports[p].size)-
                          (comp-
>nets_list[q].number))-1 );
       fprintf(verilog_file,"%d]",((temp-
>port_array[p])-(comp->nets_list[q].number)));
      }
      else if (comp->nets_list[q].size!=1)
        /* Connecting a single bit to a single bit in a
bus */
        fprintf(verilog_file,"[%d]",((temp-
>port_array[p])-(comp->nets_list[q].number)) );
      fprintf(verilog_file,")");
     }
     if (p<(temp_comp->num_ports-1))
      fprintf(verilog_file,", ");
   }
   fprintf(verilog_file,");\n");
   temp=temp->next;
   n++;
  }
  } /****** end if id *********/ else {
   m=lookup_component(temp->component_id);
   temp_comp=component_lib[m-1].comp_pointer;
   fprintf(verilog_file,"%s","assign\n");
   fprintf(verilog_file,"%s",temp_comp-
>verilog_equation);
   fprintf(verilog_file,";\n");
  }
  fprintf(verilog_file,"endmodule\n");
}

/*****************************************
******************/

void write_files(char *filename)
/* filename are the files to be written - .vhdl and .vs
are appended */


{
  char vhdl_filename[255], verilog_filename[255];
  char temp_string[255];
  unsigned
diff_comp[MAX_DIFFERENT_COMPONENTS];
  unsigned n, m, p;
  struct component_type *temp;
  FILE *vhdl_file, *verilog_file;
  component_lib_type *temp_comp;

  strcpy(vhdl_filename,filename);
  strcat(vhdl_filename,".vhdl");
  strcpy(verilog_filename,filename);
  strcat(verilog_filename,".sv");
  vhdl_file=fopen(vhdl_filename,"w");
  verilog_file=fopen(verilog_filename,"w");
  for (p=0;component_lib[p].id<1000;p++);
  for (;p<num_comp;p++)
  {
   temp_comp=component_lib[p].comp_pointer;
   if (temp_comp->nets_list!=NULL)
   {
    /* Make a list of different components used in
current design. */
    for
(n=0;n<MAX_DIFFERENT_COMPONENTS;n++)
     diff_comp[n]=0;

    temp=temp_comp->comp_list;
    while (temp!=NULL)
    {
     n=lookup_component(temp->component_id);
     m=0;
     while ((diff_comp[m]!=0) &&
(diff_comp[m]!=n))
      m++;
     if (diff_comp[m]==0) /* Ie. component not
found */
      diff_comp[m]=n;
     temp=temp->next;
    }
    write_VHDL(vhdl_file,temp_comp,diff_comp);
    write_verilog(verilog_file,temp_comp);
   }
  }
  fclose(vhdl_file);
  fclose(verilog_file);
}
```

main_gen.c

```
#include"gen_defines.h"
#include"multi_file.h"
#include<stdio.h>
#include<string.h>
#include<math.h>

char comp_name[255];
unsigned temp_comp_id=10001;
unsigned q=0;
unsigned rom_counter;
unsigned hamming_mux_counter;

component_lib_type *new_component(char *name)
{
  component_lib_type *base;

  base=new component_lib_type;
  base->nets_list=NULL;
  base->comp_list=NULL;
  base->ports=NULL;
  base->num_nets=0;
  base->num_ports=0;
  base->vhdl_equation=new
char[strlen(comp_name)];
  base->verilog_equation=new
char[strlen(comp_name)];
  base->library_name=new char[strlen(name)+1];
  strcpy(base->library_name,name);
  return base;
}


void macro_block_generation()
{
  component_lib_type *temp;
  unsigned choice, bus_size,
base_v,smallest,biggest,nbaddr,nbbits,periode,fa_typ
e,g,k;
  char rom_package[10],rom_file_name[10];
  unsigned weight[20]={0,1,2,3};
  printf("\n=========================");
  printf("\n  Macro block generation  ");
  printf("\n\n 1. Single full-adder (or inclusive)");
  printf("\n 2. Single full-adder (or exclusive)");
  printf("\n 3. Booth multiplier with reduced
decoder");
  printf("\n 6.  ROM");
  printf("\n 7.  mod_A generator using period of A");
  printf("\n 8.  mod_A generator using half period of
A");
  printf("\n 9.  multi_operand modular adder");
  printf("\n 10.  buffer block");
  printf("\n 11.  AxB modulo base using half periode
of the base");
  printf("\n 12.  Adder Fault Secure Using residue
code");
  printf("\n 13.  Translator");
  printf("\n 14.  MUX");
  printf("\n 15.  Hamming Coder");
  printf("\n 16.  Hamming multiplexer");
  printf("\n 17.  Hamming detection correction
circuit");
  printf("\n What do you want to generate : ");
  scanf("%d",&choice);

  switch (choice)
  {
   case 1 : temp=single_fadder1(comp_name);
            break;
   case 2 : temp=single_fadder2(comp_name);
            break;
   case 3 : printf("\n size of the first operand : ");
            scanf("%d",&smallest);
            printf("\n size of the second operand : ");
            scanf("%d",&biggest);
            printf("\n Full_Adder type ?
                    1. Or inclusive.
                    2. Or exclusive. ");
            scanf("%d",&fa_type);

temp=booth2_braun(comp_name,smallest,biggest,fa_
type);
            break;

   case 6 : printf("\n the number of data bits : ");
            scanf("%d",&nbaddr);
            printf("\n Base value : ");
            scanf("%d",&base_v);
            temp=rom(nbaddr,weight,base_v,0);
            break;
   case 7 : printf("\n le nombre de bits : ");
            scanf("%d",&bus_size);
            printf("\n la valeur de la base : ");
            scanf("%d",&base_v);
            printf("\n la valeur de la periode de la base :
");
            scanf("%d",&periode);

temp=mod_A_period(NULL,bus_size,base_v,periode
,1);
            break;
   case 8 : printf("\n le nombre de bits : ");
            scanf("%d",&bus_size);
            printf("\n la valeur de la base : ");
            scanf("%d",&base_v);
            printf("\n la valeur de la demi_periode de la
base : ");
            scanf("%d",&periode);

temp=mod_A_Hperiod(NULL,bus_size,base_v,perio
de,0);
            break;
   case 9 : printf("\n le nombre d'operandes : ");
            scanf("%d",&bus_size);
            printf("\n la valeur de la base : ");
            scanf("%d",&base_v);
            printf("\n la valeur de la periode de la base :
");
```

```
          scanf("%d",&periode);

temp=multi_operand_modular_adder(NULL,bus_siz
e,base_v,periode);
          break;
   case 10 : printf("\n le nombre de bits: ");
           scanf("%d",&bus_size);
           temp=bufer_block(NULL,bus_size);
           break;
   case 11 : printf("\n le nombre de bits: ");
           scanf("%d",&bus_size);

temp=A_x_B_modulo_Hperiode(NULL,bus_size);
          break;
   case 12 : printf("\n le nombre de bits: ");
           scanf("%d",&bus_size);
           temp=adder_fs_cr(NULL,bus_size);
           break;
   case 13 : printf("\n le nombre de bits: ");
           scanf("%d",&bus_size);
           temp=translator(NULL,bus_size);
           break;
   case 14 : printf("\n le nombre de bits : ");
          scanf("%d",&bus_size);
               printf("\n le nombre de bits d'adresses :
");
          scanf("%d",&nbaddr);
          printf("\n Mux type : ");
          scanf("%d",&base_v);
               printf("\n le nombre de bits de parite : ");
          scanf("%d",&g);
               printf("\n quel groupe de parite : ");
          scanf("%d",&k);
          temp=mux(bus_size,nbaddr,base_v,k,g);
          break;
   case 15 : printf("\n le nombre de bits de donnees:
");
           scanf("%d",&bus_size);
           temp=codeur_hamming(NULL,bus_size);
           break;
   case 16 : printf("\n le nombre de bits de donnees:
");
           scanf("%d",&bus_size);
           temp=hamming_mux(bus_size);
           break;
   case 17 : printf("\n le nombre de bits de donnees:
");
           scanf("%d",&bus_size);
                printf("\n le nombre de bits d'addresse:
");
           scanf("%d",&nbaddr);

temp=hamming_detection_correction(NULL,bus_siz
e,nbaddr);
          break;
   default : printf("\n Error : Choice not available
!\n");
          break;
  }
 insert_component_in_lib(temp);
}
```

```
void single_block_generation()
{
 component_lib_type *temp=NULL;
 unsigned choice,width,
width1,smallest,biggest,bus_size,fa_type,nc,nbits;
 prog *order;
 printf("\n==========================");
 printf("\n 1.  wallace multiplier");
 printf("\n 2.  Booth2 multiplier");
 printf("\n 3.  Booth2 Wallace Multiplier");
 printf("\n 4.  Modulo A=2^n-1 adder based on carry
lookahead design ");
 printf("\n 5.  Shifter ");
 printf("\n What do you want to generate : ");
 scanf("%d",&choice);

 switch (choice)
 {
 case 1 : printf("\n size of the first operand (biggest) :
");
   scanf("%d",&biggest);
   printf("\n size of the second operand : ");
   scanf("%d",&smallest);
   printf("\n which structure of full_adder cell you
want to generate
                1-FA1  2-FA2 : ");
   scanf("%d",&fa_type);

temp=mult_wallace(comp_name,smallest,biggest,fa_
type);
   break;

 case 2 : printf("\n size of the first operand (biggest) :
");
   scanf("%d",&biggest);
   printf("\n size of the second operand (smallest) : ");
   scanf("%d",&smallest);
   printf("\n which structure of full_adder cell you
want to generate
                1-FA1  2-FA2 : ");
   scanf("%d",&fa_type);

temp=booth2_braun(comp_name,biggest,smallest,fa_
type);
   break;
 case 3 : printf("\n size of the first operand (biggest) :
");
   scanf("%d",&biggest);
   printf("\n size of the second operand : ");
   scanf("%d",&smallest);
   printf("\n which structure of full_adder cell you
want to generate
                1-FA1  2-FA2 : ");
   scanf("%d",&fa_type);

temp=booth2_wallace(comp_name,biggest,smallest,f
a_type);
   break;
 case 4 : printf("\n The base value ? : ");
   scanf("%d",&bus_size);
```

```
    temp=modulo_A_adder(comp_name,bus_size);
    break;
  case 5 : printf("\n  : size of the operand ");
    scanf("%d",&bus_size);
    nc=int((int) (log10(bus_size)/log10(2)));
    order=new prog[1];

printf("\n\n*******************************");
    printf("\n 1. Rotation");
    printf("\n 2. Logic Left Shift");
    printf("\n 3. Logic Right Shift");
    printf("\n 4. Arithmetic Shift");
    printf("\nType one of the above choices to answer
the following questions");
    printf("\nFirst operation  (address 00) : ");
    scanf("%d",&order->list[0]);
    printf("\nSecond operation (address 01) : ");
    scanf("%d",&order->list[1]);
    printf("\nThird operation  (address 10) : ");
    scanf("%d",&order->list[2]);
    printf("\nFourth operation (address 11) : ");
    scanf("%d",&order->list[3]);
    temp=shifter(NULL,bus_size,3,1,nc,2,order);;
    break;
  default : printf("\n Error : Choice not available !\n");
    break;
  }
  insert_component_in_lib(temp);
}

void system_block_generation()
{
  component_lib_type *temp=NULL;
  unsigned choice,width,
width1,smallest,biggest,fa_type,bus_size;

  printf("\n=========================");
  printf("\n 1.  Fault Secure (residue code) Wallace
multiplier");
  printf("\n 2.  Booth2 multiplier fault secure");
  printf("\n 3.  Booth2 multiplier with double
recoder");
  printf("\n 4. Adder Fault Secure Based On Code
Residue");
  printf("\n 5. Wallace Multiplier Checked by Several
Groups of Parity ");
  printf("\n 6. Booth-Wallace Multiplier Checked by
Several Groups of Parity");
  printf("\n 7. Adder Fault Secure Checked by Several
Groups of Parity");
  printf("\n 8.  Fault Secure (residue code)
Booth_Wallace multiplier");
  printf("\n 9.  Fault Secure (Parity Prediction) Shifter
with two bits of Parity");
  printf("\n What do you want to generate : ");
  scanf("%d",&choice);

  switch (choice)
  {
  case 1 : printf("\n size of the first operand (biggest) :
");
    scanf("%d",&biggest);
    printf("\n size of the second operand (smallest) : ");
    scanf("%d",&smallest);

temp=wallace_fs_cr(comp_name,smallest,biggest);
    break;
  case 2 : printf("\n size of the first operand (biggest) :
");
    scanf("%d",&biggest);
    printf("\n size of the second operand (smallest) : ");
    scanf("%d",&smallest);

temp=multiplier_booth_braun_fs_cr(comp_name,sm
allest,biggest);
    break;
  case 3 : printf("\n size of the first operand (biggest) :
");
    scanf("%d",&biggest);
    printf("\n size of the second operand (smallest) : ");
    scanf("%d",&smallest);
    printf("\n which structure of full_adder cell you
want to generate
                1-FA1  2-FA2 : ");
    scanf("%d",&fa_type);

temp=booth2_braun_double_recoder(comp_name,big
gest,smallest,fa_type);
    break;
  case 4 : printf("\n Width of operands ?\n");
    scanf("%d",&bus_size);
    temp=adder_fs_cr(comp_name,bus_size);
    break;
  case 5 : printf("\n size of the first operand (biggest) :
");
    scanf("%d",&biggest);
    printf("\n size of the second operand (smallest) : ");
    scanf("%d",&smallest);
    printf("\n which structure of full_adder cell you
want to generate
                1-FA1  2-FA2 : ");
    scanf("%d",&fa_type);

temp=mult_wallace_parity_predict(comp_name,smal
lest,biggest,fa_type);
    break;
  case 6 : printf("\n size of the first operand (biggest) :
");
    scanf("%d",&biggest);
    printf("\n size of the second operand (smallest) : ");
    scanf("%d",&smallest);
    printf("\n which structure of full_adder cell you
want to generate
                1-FA1  2-FA2 : ");
    scanf("%d",&fa_type);

temp=pc_booth2_wallace(comp_name,biggest,smalle
st,fa_type);
    break;
  case 7 : printf("\n Width of operands ? : ");
    scanf("%d",&bus_size);
```

```
temp=adder_fs_with_groups_of_parity(comp_name,
bus_size);
    break;
  case 8 : printf("\n size of the first operand (biggest) :
");
    scanf("%d",&biggest);
    printf("\n size of the second operand (smallest) : ");
    scanf("%d",&smallest);
    temp=bo_wa_fs_cr(comp_name,smallest,biggest);
    break;
  case 9 : printf("\n size of the operand : ");
    scanf("%d",&biggest);
    temp=sc_shifter(comp_name,biggest);
    break;
  default : printf("\n Error : Choice not available !\n");
    break;
  }
  insert_component_in_lib(temp);
}

main()
{
  char filename[255];
  int menu_choice;
  rom_counter=1;
  hamming_mux_counter=1;
  printf("\n macro-cell generator - version II\n");
  printf("\nFilename for output (*.sv and *.vhdl) : ");
  scanf("%s",filename);
  printf("\nName of entity/module : ");
  scanf("%s",comp_name);
  printf("\n\n**************************");
  printf("\n 1. Macro Block");
  printf("\n 2. Single Block ");
  printf("\n 3. Single Block Fault Secure");
  printf("\nWhat do you want to generate : ");
  scanf("%d",&menu_choice);
  insert_equation_component_in_lib();
  switch (menu_choice)
  {
    case 1 : macro_block_generation();
             break;
    case 2 : single_block_generation();
             break;
    case 3 : system_block_generation();
             break;
    default : printf("\nError : Choice not available
!\n");
              break;
  }
  write_files(filename);

}
```

Generateur modulo A utilisant la demi_periode de A.
Dans ce programme pA veut dire la demi periode de A.

```
component_lib_type *mod_A_Hperiod(char *name,
unsigned nbits,unsigned A,unsigned pA,unsigned
two_comp)
{
  component_lib_type *base,*temp;
  unsigned d=(int) (ceil(log10(A)/log10(2)));
  net_tuple io_list[MAX_NETS],
io_list1[MAX_NETS],buffer_bus[20];
  net_tuple fil[pA], fil_c[pA], fil_suiv,fil_suiv_c,fil_r;
  net_tuple
bus_pos[pA],bus_int[pA],bus_rom[d],bus_befor_resu
lt,bus_buffer,wire,wire1,wire2,residue_inverted;
  unsigned ind[pA],q[pA];
  unsigned c1[pA],c2[pA];
  unsigned nb1[pA],nb2[pA];
  unsigned repere[pA];
  unsigned
weight[30],weight1[30],cor_counter=0,correction;
  int i,j,Nt,dec,l,k,count_bits,compteur=0;

  if (name==NULL)
    {
     char temp_string[255];

sprintf(temp_string,"Mod_%d_Gen_Half_Period_%d
_INPUT",A,nbits);
     base=new_component(temp_string);
    }
  else
    base=new_component(name);
  base->id=temp_comp_id;
  temp_comp_id++;
  build_external_port(base,"X",IN,nbits);
  build_external_port(base,"XmodA",OUT,d);
  Nt=nbits;
  for(i=0;i<pA;i++) {
    nb1[i]=0;
    nb2[i]=0;
    ind[i]=0;
    c1[i]=0;
    c2[i]=0;
  }
  for(i=0;i<30;i++) {
    weight[i]=0;
    weight1[i]=0;
  }
  while(Nt>2*pA) {
    if(Nt==nbits) {/*** La premiere etape ***/
      /* Calcul des nb1[i] pour la premiere etape ou
            nb1[i] est le nbr des entrees qui ont la meme
            valeur de residu modulo A */
      for(i=0;i<pA;i++) {
           k=i;
           while(k<nbits) {
             nb1[i]++;
             k=k+pA;
           }
      }

      /* Calcul du nombre de sorties de meme poids
nb2[i] */
      for(i=0;i<pA;i++) {
           if(((nb1[i]%3)==0)||((nb1[i]%3)==2)) {
             nb2[i]=(nb1[i]+1)/3+nb2[i];

nb2[(i+1)%pA]=(nb1[i]+1)/3+nb2[(i+1)%pA];
           }else {
             nb2[i]=nb1[i]/3+1+nb2[i];
             nb2[(i+1)%pA]=nb1[i]/3+nb2[(i+1)%pA];
           }
      }
      /* allocation des bus_pos[i] */
      for(i=0;i<pA;i++)
           bus_pos[i]=allocate_net(base,nb2[i]);
      /* Instans of components */
      for(i=0;i<pA;i++) {
           for(j=0;j<nb1[i]/3;j++) {
             for(l=0;l<3;l++) {

if((i+j*3*pA+l*pA)/pA==2*((i+j*3*pA+l*pA)/2*pA
)) {
                  if((i+j*3*pA+l*pA==nbits-
1)&&(two_comp)){ /*le bit de signe (on inverse si
(nbits-1)/pA est pair)*/
                       wire=allocate_net(base,1);
                       io_list1[0]=get_port(base,1);
                       io_list1[0].size=1;

           io_list1[0].number+=(i+j*3*pA+l*pA);
                       io_list1[1]=wire;

           instans_of_component(base,INVERTER,io_
list1);
                       io_list[l]=wire;
                  }else {
                       io_list[l]=get_port(base,1);
                       io_list[l].size=1;

           io_list[l].number+=(i+j*3*pA+l*pA);
                  }
                 }else {
                  if((i+j*3*pA+l*pA==nbits-
1)&&(two_comp)){
                       io_list[l]=get_port(base,1);
                       io_list[l].size=1;

           io_list[l].number+=(i+j*3*pA+l*pA);
                  }else {
                       wire=allocate_net(base,1);
                       io_list1[0]=get_port(base,1);
                       io_list1[0].size=1;

           io_list1[0].number+=(i+j*3*pA+l*pA);
                       io_list1[1]=wire;
```

```
        instans_of_component(base,INVERTER,io_
list1);
                io_list[l]=wire;
              }
             }
            }
            io_list[3]=bus_pos[(i+1)%pA];
            io_list[3].size=1;
            io_list[3].number+=c2[(i+1)%pA];
            c2[(i+1)%pA]++;
            io_list[4]=bus_pos[i];
            io_list[4].size=1;
            io_list[4].number+=c2[i];
            c2[i]++;
             if(i==pA-1) {

instans_of_component(base,SINGLE_FADDER2_CI
,io_list);
               cor_counter++;
             }
             else

instans_of_component(base,SINGLE_FADDER2,io_
list);
          }
          if((nb1[i]%3)==1) {
            io_list[0]=get_port(base,1);
            io_list[0].size=1;
            io_list[0].number+=i+(nb1[i]/3)*3*pA;
            io_list[1]=bus_pos[i];
            io_list[1].size=1;
            io_list[1].number+=c2[i];
            c2[i]++;
            if(i+(nb1[i]/3)*3*pA==nbits-1) {
              if(((two_comp==1)&&((nbits-
1)%2==0))||((two_comp==0)&&((nbits-1)%2==1)))

instans_of_component(base,INVERTER,io_list);
              else

instans_of_component(base,BUFER,io_list);
            }else {
             if((i+(nb1[i]/3)*3*pA)%2==0)

instans_of_component(base,BUFER,io_list);
              else

instans_of_component(base,INVERTER,io_list);
            }
          }
          if((nb1[i]%3)==2) {
            io_list[0]=get_port(base,1);
            io_list[0].size=1;
            io_list[0].number+=(i+(nb1[i]/3)*3*pA);
            io_list1[0]=get_port(base,1);
            io_list1[0].size=1;

io_list1[0].number+=(i+(nb1[i]/3)*3*pA+pA);
            io_list1[1]=allocate_net(base,1);
            if(i+(nb1[i]/3)*3*pA+pA==nbits-1) {
              if(((two_comp==1)&&((nbits-
1)%2==0))||((two_comp==0)&&((nbits-1)%2==1)))

instans_of_component(base,INVERTER,io_list1);
              else

instans_of_component(base,BUFER,io_list1);
            }else {
             if((i+(nb1[i]/3)*3*pA+pA)%2==0)

instans_of_component(base,BUFER,io_list1);
              else

instans_of_component(base,INVERTER,io_list1);
            }
            io_list[1]=io_list1[1];
            io_list[2]=bus_pos[i];
            io_list[2].size=1;
            io_list[2].number+=c2[i];
            c2[i]++;
            io_list[3]=bus_pos[(i+1)%pA];
            io_list[3].size=1;
            io_list[3].number+=c2[(i+1)%pA];
            c2[(i+1)%pA]++;
            if(i==pA-1)

instans_of_component(base,SINGLE_HADDER_CI,
io_list);
            else

instans_of_component(base,SINGLE_HADDER,io_l
ist);
          }
      }
    }else {
      /****Nt est different de nbits ********/
      for(i=0;i<pA;i++) {
            nb1[i]=nb2[i];
            nb2[i]=0;
            bus_int[i]=bus_pos[i];
            c1[i]=0;
            c2[i]=0;
      }
      /***** Calcul du nombre de sorties de meme
poids nb2[i] *****/
      for(i=0;i<pA;i++) {
            if(((nb1[i]%3)==0)||((nb1[i]%3)==2)) {
              nb2[i]=(int) ((nb1[i]+1)/3)+nb2[i];
              nb2[((i+1)%pA)]=(int)
((nb1[i]+1)/3)+nb2[((i+1)%pA)];
            }else {
              nb2[i]=(int) ((nb1[i])/3)+1+nb2[i];
              nb2[(i+1)%pA]=(int)
((nb1[i])/3)+nb2[(i+1)%pA];
            }
      }
      /* allocation des bus_pos[i] */
      for(i=0;i<pA;i++)
            bus_pos[i]=allocate_net(base,nb2[i]);
```

```
    /********** Instans of component
**************/
     for(i=0;i<pA;i++) {
      for(j=0;j<nb1[i]/3;j++) {
            for(l=0;l<3;l++) {
              io_list[l]=bus_int[i];
              io_list[l].size=1;
              io_list[l].number+=c1[i];
              c1[i]++;
            }
            io_list[3]=bus_pos[(i+1)%pA];
            io_list[3].size=1;
            io_list[3].number+=c2[(i+1)%pA];
            c2[(i+1)%pA]++;
            io_list[4]=bus_pos[i];
            io_list[4].size=1;
            io_list[4].number+=c2[i];
            c2[i]++;
            if(i==pA-1) {

instans_of_component(base,SINGLE_FADDER2_CI
,io_list);
              cor_counter++;
            }
            else

instans_of_component(base,SINGLE_FADDER2,io_
list);
       }
      io_list[0]=bus_int[i];
      io_list[0].size=1;
      io_list[0].number+=c1[i];
      c1[i]++;
      if((nb1[i]%3)==2) {
            io_list[1]=bus_int[i];
            io_list[1].size=1;
            io_list[1].number+=c1[i];
            c1[i]++;
            io_list[2]=bus_pos[i];
            io_list[2].size=1;
            io_list[2].number+=c2[i];
            c2[i]++;
            io_list[3]=bus_pos[(i+1)%pA];
            io_list[3].size=1;
            io_list[3].number+=c2[(i+1)%pA];
            c2[(i+1)%pA]++;
            if(i==pA-1) {

instans_of_component(base,SINGLE_HADDER_CI,
io_list);
              cor_counter++;
            }
            else

instans_of_component(base,SINGLE_HADDER,io_l
ist);
       }
      if(nb1[i]%3==1){
            io_list[1]=bus_pos[i];
            io_list[1].size=1;
            io_list[1].number+=c2[i];
              c2[i]++;

instans_of_component(base,BUFER,io_list);
       }
      }
    }/*end Nt different de nbits */
    compteur=compteur+1;
    Nt=0;
    for(i=0;i<pA;i++)
      Nt=Nt+nb2[i];
  }/*end while*/
  printf("cor_counter=%d\n",cor_counter);

  /* Derniere etape pour obtenir B mod A */
  if(A==(int)(pow(2.0,d-1)+1)) { /*A est de la forme
2**a+1*/
    bus_befor_result=allocate_net(base,d);
    if((nbits<=2*pA)&&(nbits>pA)) {
      for(i=0;i<pA;i++) {
            if(i+pA<nbits) {
              io_list[0]=get_port(base,1);
              io_list[0].size=1;
              io_list[0].number+=i;
              wire=allocate_net(base,1);
              io_list1[0]=get_port(base,1);
              io_list1[0].size=1;
              io_list1[0].number+=i+pA;
              io_list1[1]=wire;

instans_of_component(base,INVERTER,io_list1);
              io_list[1]=wire;
              if(i==0) {
                io_list[2]=bus_befor_result;
                io_list[2].size=1;
                io_list[2].number+=0;
              }else {
                fil_suiv_c=fil_suiv;
                io_list[2]=fil_suiv_c;
              }
              if(i==pA-1) {
                io_list[3]=bus_befor_result;
                io_list[3].size=1;
                io_list[3].number+=pA;
              }else {
                fil_suiv=allocate_net(base,1);
                io_list[3]=fil_suiv;
              }
              if(i!=0) {
                io_list[4]=bus_befor_result;
                io_list[4].size=1;
                io_list[4].number+=i;

instans_of_component(base,SINGLE_FADDER2,io_
list);
              }else

instans_of_component(base,SINGLE_HADDER,io_l
ist);
            }else {
              io_list[0]=get_port(base,1);
              io_list[0].size=1;
```

```
            io_list[0].number+=i;
            io_list[1]=fil_suiv;
            io_list[2]=bus_befor_result;
            io_list[2].size=1;
            io_list[2].number+=i;
            if(i==pA-1) {
              io_list[3]=bus_befor_result;
              io_list[3].size=1;
              io_list[3].number+=pA;
            }else {
              fil_suiv=allocate_net(base,1);
              io_list[3]=fil_suiv;
            }

instans_of_component(base,SINGLE_HADDER,io_l
ist);
          }
      }
    }else {
      for(i=0;i<pA;i++) {
          io_list[0]=bus_pos[i];
          io_list[0].size=1;
          io_list[0].number+=0;
          io_list[1]=bus_pos[i];
          io_list[1].size=1;
          io_list[1].number+=1;
          if(i==0) {
            io_list[2]=bus_befor_result;
            io_list[2].size=1;
            io_list[2].number+=0;
          }else
            io_list[2]=fil_suiv;
          if(i==pA-1) {
            io_list[3]=bus_befor_result;
            io_list[3].size=1;
            io_list[3].number+=pA;
          }else {
            fil_suiv=allocate_net(base,1);
            io_list[3]=fil_suiv;
          }
          if(i!=0) {
            io_list[4]=bus_befor_result;
            io_list[4].size=1;
            io_list[4].number+=i;

instans_of_component(base,SINGLE_FADDER2,io_
list);
          }else

instans_of_component(base,SINGLE_HADDER,io_l
ist);
      }
    }
    /*la rom donnant le resultat final*/
    for(i=0;i<d;i++) weight1[i]=i;
    correction=(-
cor_counter+2*(int)(ceil(nbits/pA)/2))%A;
    printf("correction=%d",correction);
    temp=rom(d,weight1,A,correction);
    insert_component_in_lib(temp);
    io_list[0]=bus_befor_result;
    io_list[1]=get_port(base,2);
    instans_of_component(base,temp->id,io_list);
  }/*end A est de la forme 2**a+1 */

  else { /*A est different de la forme 2**a+1 */
    if(nbits<=2*pA) {
      if(nbits<pA+d) {
          if(nbits<=7) {
            for(i=0;i<nbits;i++)
              weight[i]=i;
            temp=rom(nbits,weight,A,0);
            insert_component_in_lib(temp);
            io_list[0]=get_port(base,1);
            io_list[1]=get_port(base,2);
            instans_of_component(base,temp-
>id,io_list);
          }else { /*nbits >7*/
            for(i=0;i<((int)(nbits-d)/5);i++) {
              bus_rom[i]=allocate_net(base,d);
              if(i==(int)((nbits-d)/5)-1) {/*la derniere
iteration*/
                if((nbits-d)%5==0) {/*une seule rom a la
fin*/
                    for(j=nbits-5;j<nbits;j++)
                      weight[j-(nbits-5)]=j;
                    temp=rom(5,weight,A,0);
                    insert_component_in_lib(temp);
                    io_list[0]=get_port(base,1);
                    io_list[0].size=5;
                    io_list[0].number+=(nbits-5);
                    io_list[1]=bus_rom[i];
              instans_of_component(base,temp-
>id,io_list);
                }else {
                    if((nbits-d)%5<=2){/*une seule
rom a la fin*/
                      dec=(nbits-d)%5;
                      for(j=nbits-5-dec;j<nbits;j++)
                        weight[j-(nbits-5-dec)]=j;
                      temp=rom(5+dec,weight,A,0);
                      insert_component_in_lib(temp);
                      io_list[0]=get_port(base,1);
                      io_list[0].size=5+dec;
                      io_list[0].number+=(nbits-5-dec);
                      io_list[1]=bus_rom[i];
                      instans_of_component(base,temp-
>id,io_list);
                    }else {/*il y a deux roms a la fin*/
                      dec=(nbits-d)%5;
                      for(j=nbits-5-dec;j<nbits-dec;j++)
                        weight[j-(nbits-5-dec)]=j;
                      temp=rom(5,weight,A,0);
                      insert_component_in_lib(temp);
                      io_list[0]=get_port(base,1);
                      io_list[0].size=5;
                      io_list[0].number+=(nbits-5-dec);
                      io_list[1]=bus_rom[i];
                      instans_of_component(base,temp-
>id,io_list);
                      for(j=nbits-dec;j<nbits;j++)
                        weight[j-(nbits-dec)]=j;
```

```
                    temp=rom(dec,weight,A,0);
                    insert_component_in_lib(temp);
                    io_list[0]=get_port(base,1);
                    io_list[0].size=dec;
                    io_list[0].number+=(nbits-dec);
                    io_list[1]=bus_rom[i+1];
                    instans_of_component(base,temp-
>id,io_list);
                  }
                }

              }else {/*les autres iterations*/
                for(j=i*5+d;j<i*5+d+5;j++)
                      weight[j-(i*5+d)]=j;
                temp=rom(5,weight,A,0);
                insert_component_in_lib(temp);
                io_list[0]=get_port(base,1);
                io_list[0].size=5;
                io_list[0].number+=(i*5+d);
                io_list[1]=bus_rom[i];
                instans_of_component(base,temp-
>id,io_list);
              }
            }
            k=(int)((nbits-d)/5)+1;
            if((nbits-d)%5>2) k++;

temp=multi_operand_modular_adder(NULL,k,A,pA)
;
            insert_component_in_lib(temp);
            io_list[0]=get_port(base,1);
            io_list[0].size=d;
            io_list[0].number+=0;
            for(i=1;i<k;i++) io_list[i]=bus_rom[i-1];
            l=(int)(ceil(log10((int)(k*(A-
1)+1))/log10(2)));
            bus_befor_result=allocate_net(base,l);
            io_list[k]=bus_befor_result;
            instans_of_component(base,temp-
>id,io_list);
            /*la rom donnant le resultat final*/
            for(i=0;i<l;i++) weight1[i]=i;
            temp=rom(l,weight1,A,0);
            insert_component_in_lib(temp);
            io_list[0]=bus_befor_result;
            io_list[1]=get_port(base,2);
            instans_of_component(base,temp-
>id,io_list);
          }
      }/*end nbits<pA+d*/
      else {/*nbits>=pA+d*/
          printf("right\n");
          temp=bufer_block(NULL,nbits-2*d);
          insert_component_in_lib(temp);
          count_bits=0;
          for(i=0;i<nbits;i++) {
            if(((i>=d)&&(i<pA))||(i>=pA+d)) {
              io_list[count_bits]=get_port(base,1);
              io_list[count_bits].size=1;
              io_list[count_bits].number+=i;
              weight[count_bits]=i;
              count_bits++;
            }
          }
          bus_buffer=allocate_net(base,nbits-2*d);
          io_list[nbits-2*d]=bus_buffer;
          instans_of_component(base,temp-
>id,io_list);
          printf("right\n");
          printf("count_bits=%d\n",count_bits);

          for(i=0;i<((int)(nbits-2*d)/5);i++) {
            bus_rom[i]=allocate_net(base,d);
            if(i==(int)((nbits-2*d)/5)-1) {/*la derniere
iteration*/
              if((nbits-2*d)%5==0) {/*une seule rom a
la fin*/
                for(j=0;j<5;j++) {
                    weight1[j]=weight[count_bits-5+j];
                }
                temp=rom(5,weight1,A,0);
                insert_component_in_lib(temp);
                io_list[0]=bus_buffer;
                io_list[0].size=5;
                io_list[0].number+=(count_bits-5);
                io_list[1]=bus_rom[i];
                instans_of_component(base,temp-
>id,io_list);
              }else {
                if(count_bits%5<=2){/*une seule rom a
la fin*/
                    dec=(count_bits)%5;
                    for(j=0;j<5+dec;j++)
                      weight1[j]=weight[count_bits-5-
dec+j];
                    temp=rom(5+dec,weight1,A,0);
                    insert_component_in_lib(temp);
                    io_list[0]=bus_buffer;
                    io_list[0].size=5+dec;
                    io_list[0].number+=(count_bits-5-
dec);
                    io_list[1]=bus_rom[i];
                    instans_of_component(base,temp-
>id,io_list);
                }else {/*il y a deux rom a la fin*/
                    dec=(count_bits)%5;
                    for(j=0;j<5;j++)
                      weight1[j]=weight[count_bits-5-
dec+j];
                    temp=rom(5,weight1,A,0);
                    insert_component_in_lib(temp);
                    io_list[0]=bus_buffer;
                    io_list[0].size=5;
                    io_list[0].number+=(nbits-5-dec);
                    io_list[1]=bus_rom[i];
                    instans_of_component(base,temp-
>id,io_list);
                    for(j=0;j<dec;j++)
                      weight1[j]=weight[count_bits-
dec+j];
                    temp=rom(dec,weight1,A,0);
                    insert_component_in_lib(temp);
```

```
                io_list[0]=bus_buffer;
                io_list[0].size=dec;
                io_list[0].number+=(count_bits-
dec);
                bus_rom[i+1]=allocate_net(base,d);
                io_list[1]=bus_rom[i+1];
                instans_of_component(base,temp-
>id,io_list);
              }
             }

            }else {/*les autres iterations*/
              printf("right\n");
              for(j=0;j<5;j++)
                weight1[j]=weight[i*5+j];
              temp=rom(5,weight1,A,0);
              insert_component_in_lib(temp);
              io_list[0]=bus_buffer;
              io_list[0].size=5;
              io_list[0].number+=(i*5);
              io_list[1]=bus_rom[i];
              instans_of_component(base,temp-
>id,io_list);
            }
          }
          printf("right\n");
          io_list1[0]=get_port(base,1);
          io_list1[0].size=d;
          io_list1[0].number+=pA;
          residue_inverted=allocate_net(base,d);
          io_list1[1]=residue_inverted;
          temp=inverter_block(NULL,d);
          insert_component_in_lib(temp);
          instans_of_component(base,temp-
>id,io_list1);

          k=(int)((nbits-2*d)/5)+2;
          if((nbits-2*d)%5>2) k++;
          temp=multi_operand_modular_adder(NULL
,k,A,pA);
          insert_component_in_lib(temp);
          io_list[0]=get_port(base,1);
          io_list[0].size=d;
          io_list[0].number+=0;
          io_list[1]=residue_inverted;

          for(i=2;i<k;i++) io_list[i]=bus_rom[i-2];
          l=(int)(ceil(log10((int)(k*(A-
1)+1))/log10(2)));
          bus_befor_result=allocate_net(base,l);
          io_list[k]=bus_befor_result;
          instans_of_component(base,temp-
>id,io_list);
          /*la rom donnant le resultat final*/
          for(i=0;i<l;i++) weight1[i]=i;
          correction=(int)(pow(2.0,d)-1)%A;
          printf("correction=%d",correction);
          temp=rom(l,weight1,A,correction);
          insert_component_in_lib(temp);
          io_list[0]=bus_befor_result;
          io_list[1]=get_port(base,2);
          instans_of_component(base,temp-
>id,io_list);
        }/*end nbits>=pA+d*/

      }/*end nbits <=2*pA*/
      else {/*nbits>2*pA*/
        /*les 2 premiers residues*/
        temp=bufer_block(NULL,d);
        insert_component_in_lib(temp);
        for(i=0;i<2;i++) {
          for(j=0;j<d;j++) {
            io_list[j]=bus_pos[j];
            io_list[j].size=1;
            io_list[j].number+=i;
          }
          buffer_bus[i]=allocate_net(base,d);
          io_list[d]=buffer_bus[i];
          instans_of_component(base,temp-
>id,io_list);
        }
       /*les bits restant assembles dans un seul bus:
bus_buffer*/
        for(i=0;i<2*(pA-d);i++) {
          io_list[i]=bus_pos[d+i/2];
          io_list[i].size=1;
          io_list[i].number+=(i%2);
          weight[i]=d+i/2;
        }
        bus_buffer=allocate_net(base,2*(pA-d));
        io_list[2*(pA-d)]=bus_buffer;
        if(d==2*(pA-d))
          instans_of_component(base,temp-
>id,io_list);
        else {
          temp=bufer_block(NULL,2*(pA-d));
          insert_component_in_lib(temp);
          instans_of_component(base,temp-
>id,io_list);
        }

        count_bits=2*(pA-d);
        /*les roms correspondant aux bits restant*/
        for(i=0;i<((int)(count_bits/5));i++) {
          bus_rom[i]=allocate_net(base,d);
          if(i==(int)(count_bits/5)-1) {/*la derniere
iteration*/
            if(count_bits%5==0) {/*une seule rom a la
fin*/
              for(j=0;j<5;j++)
                weight1[j]=weight[count_bits-5+j];
              temp=rom(5,weight1,A,0);
              insert_component_in_lib(temp);
              io_list[0]=bus_buffer;
              io_list[0].size=5;
              io_list[0].number+=(count_bits-5);
              io_list[1]=bus_rom[i];
              instans_of_component(base,temp-
>id,io_list);
            }else {
              if(count_bits%5<=2){/*une seule rom a la
fin*/
```

```
                dec=(count_bits)%5;
                for(j=0;j<5+dec;j++)
                        weight1[j]=weight[count_bits-5-
dec+j];
                temp=rom(5+dec,weight1,A,0);
                insert_component_in_lib(temp);
                io_list[0]=bus_buffer;
                io_list[0].size=5+dec;
                io_list[0].number+=(count_bits-5-dec);
                io_list[1]=bus_rom[i];
                instans_of_component(base,temp-
>id,io_list);
              }else {/*il y a deux rom a la fin*/
                dec=(count_bits)%5;
                for(j=0;j<5;j++)
                        weight1[j]=weight[count_bits-5-
dec+j];
                temp=rom(5,weight1,A,0);
                insert_component_in_lib(temp);
                io_list[0]=bus_buffer;
                io_list[0].size=5;
                io_list[0].number+=(count_bits-5-dec);
                io_list[1]=bus_rom[i];
                instans_of_component(base,temp-
>id,io_list);
                for(j=0;j<dec;j++)
                        weight1[j]=weight[count_bits-
dec+j];
                temp=rom(dec,weight1,A,0);
                insert_component_in_lib(temp);
                io_list[0]=bus_buffer;
                io_list[0].size=dec;
                io_list[0].number+=(count_bits-dec);
                bus_rom[i+1]=allocate_net(base,d);
                io_list[1]=bus_rom[i+1];
                instans_of_component(base,temp-
>id,io_list);
               }
              }

            }else {/*les autres iterations*/
              printf("right\n");
              for(j=0;j<5;j++)
                weight1[j]=weight[i*5+j];
              temp=rom(5,weight1,A,0);
              insert_component_in_lib(temp);
              io_list[0]=bus_buffer;
              io_list[0].size=5;
              io_list[0].number+=(i*5);
              io_list[1]=bus_rom[i];
              instans_of_component(base,temp-
>id,io_list);
            }
      }
      printf("right\n");

      k=(int)(count_bits/5)+2;
      if(count_bits%5>2) k++;

temp=multi_operand_modular_adder(NULL,k,A,pA)
;
      insert_component_in_lib(temp);
      io_list[0]=buffer_bus[0];
      io_list[1]=buffer_bus[1];
      for(i=2;i<k;i++) io_list[i]=bus_rom[i-2];
      l=(int)(ceil(log10((int)(k*(A-1)+1))/log10(2)));
      bus_befor_result=allocate_net(base,l);
      io_list[k]=bus_befor_result;
      instans_of_component(base,temp->id,io_list);
      /*la rom donnant le resultat final*/
      for(i=0;i<l;i++) weight1[i]=i;
      correction=(-
cor_counter+2*(int)(ceil(nbits/pA)/2))%A;
      printf("correction=%d",correction);
      temp=rom(l,weight1,A,correction);
      insert_component_in_lib(temp);
      io_list[0]=bus_befor_result;
      io_list[1]=get_port(base,2);
      instans_of_component(base,temp->id,io_list);
    }
  }/*end A!=2**a+1*/

  return base;
}
```

Additionneur sûr en présence de fautes avec plusieurs bits de parité.

```
#include"gen_defines.h"
#include"multi_file.h"
#include<stdio.h>
#include<string.h>
#include<math.h>

unsigned cas=1;

component_lib_type *cellule_brent_kung1(char
*name)
{
 component_lib_type *base;
 net_tuple io_list[MAX_NETS];

 base=new_component(name);
 base->id=CELLULE_BRENT_KUNG1;

 build_external_port(base,"PI1",IN,1);
 build_external_port(base,"PI2",IN,1);
 build_external_port(base,"GAMMA1",IN,1);
 build_external_port(base,"GAMMA2",IN,1);
 build_external_port(base,"PI_BAR",OUT,1);

build_external_port(base,"GAMMA_BAR",OUT,1);

 io_list[0]=get_port(base,3);
 io_list[1]=get_port(base,2);
 io_list[2]=get_port(base,4);
 io_list[3]=get_port(base,6);

instans_of_component(base,AND2_NOR2_CELL,io
_list);

 io_list[0]=get_port(base,1);
 io_list[1]=get_port(base,2);
 io_list[2]=get_port(base,5);

instans_of_component(base,NAND2_CELL,io_list);
 return base;
}
component_lib_type *cellule_brent_kung2(char
*name)
{
 component_lib_type *base;
 net_tuple io_list[MAX_NETS];

 base=new_component(name);
 base->id=CELLULE_BRENT_KUNG2;

 build_external_port(base,"PI1_BAR",IN,1);
 build_external_port(base,"PI2_BAR",IN,1);
 build_external_port(base,"GAMMA1_BAR",IN,1);
 build_external_port(base,"GAMMA2_BAR",IN,1);
 build_external_port(base,"PI",OUT,1);
 build_external_port(base,"GAMMA",OUT,1);

 io_list[0]=get_port(base,3);
 io_list[1]=get_port(base,2);
 io_list[2]=get_port(base,4);
 io_list[3]=get_port(base,6);

instans_of_component(base,OR2_NAND2_CELL,io
_list);
 io_list[0]=get_port(base,1);
 io_list[1]=get_port(base,2);
 io_list[2]=get_port(base,5);
 instans_of_component(base,NOR2_CELL,io_list);
 return base;
}

component_lib_type *n_sklanski(char
*name,unsigned nbits)
{
 component_lib_type *base;
 net_tuple
io_list[MAX_NETS],io_list1[MAX_NETS];
 net_tuple p,p_i;
 net_tuple g,g_i;
 net_tuple pi_pos, gamma_pos;
 int i,j,k,u,marque_entree,marque_sortie,compt;
 unsigned d=(int) ceil(log10(nbits)/log10(2));
 net_tuple pi[d],gamma[d];
 int q[d],flag_sortie,flag_entree,flag_ant[d];
 if (name==NULL)
 {
  char temp_string[255];
  sprintf(temp_string,"Sklanski_%d_INPUT",nbits);
  base=new_component(temp_string);
 }
 else
  base=new_component(name);
 base->id=temp_comp_id;
 temp_comp_id++;

 build_external_port(base,"P_SMALL",IN,nbits);
 build_external_port(base,"G_SMALL",IN,nbits);
 build_external_port(base,"p_BIG",OUT,nbits);
 build_external_port(base,"G_BIG",OUT,nbits);


 io_list[0]=get_port(base,1);
 io_list[0].size=1;
 io_list[0].number+=0;
 io_list[1]=get_port(base,3);
 io_list[1].size=1;
 io_list[1].number+=0;
 instans_of_component(base,INVERTER,io_list);

 io_list[0]=get_port(base,2);
 io_list[0].size=1;
 io_list[0].number+=0;
 io_list[1]=get_port(base,4);
 io_list[1].size=1;
 io_list[1].number+=0;
 instans_of_component(base,INVERTER,io_list);

 for(i=1;i<nbits;i++) {
/*distribution des cellules sur une colonne*/
```

```
  k=i;
  for(j=0;j<d;j++) {
   q[j]=k%2;
   k=k/2;
  }

  /*****************************/
  /*realisation d'une colonne*/
  /*****************************/

  /*les entrees des cellules d'une colonne */
  for(j=0;j<d;j++) {
   if(q[j]==0) continue;
   flag_entree=0;
   flag_sortie=0;

   /* calcul du flag_entree pout la celule j*/
   for(k=j-1;k>-1;k--) {
    if(q[k]==1) {
    flag_entree=1;
    break;
    }
   }
   marque_entree=k;

   /*calcul du flag_sortie pour la cellule j*/
   for(k=j+1;k<d;k++) {
    if(q[k]==1) {
    flag_sortie=1;
    break;
    }
   }
   marque_sortie=k;
   if((i-(int)(pow(2.0,j+1)-
1))%(int)(pow(2.0,j+2))==0) {

if(((flag_sortie==0)&&(j%2==1))||(flag_sortie==1))
       flag_ant[j]=1;
      else
       flag_ant[j]=0;
   }
   /* premiere ligne */
   if((j==0)&&(q[0]==1)) {
    io_list[0]=get_port(base,1);
    io_list[0].size=1;
    io_list[0].number+=i-1;
    io_list[1]=get_port(base,1);
    io_list[1].size=1;
    io_list[1].number+=i;
    io_list[2]=get_port(base,2);
    io_list[2].size=1;
    io_list[2].number+=i-1;
    io_list[3]=get_port(base,2);
    io_list[3].size=1;
    io_list[3].number+=i;
    /* les autres lignes */
   }else {
    if(flag_ant[j-1]==1) {
    io_list[0]=pi[j-1];
    io_list[2]=gamma[j-1];
    }else {
     u=(int) (log10(i)/log10(2));
     io_list[0]=get_port(base,3);
     io_list[0].size=1;
     io_list[0].number+=((int)pow(2.0,u)-1);
     io_list[2]=get_port(base,4);
     io_list[2].size=1;
     io_list[2].number+=((int)pow(2.0,u)-1);
     }
     if(flag_entree==1) {
      if((j-marque_entree)%2==1) {
       if((i-(int)(pow(2.0,marque_entree+1)-
1))%(int)(pow(2.0,marque_entree+2))==0) {
        io_list[1]=pi[marque_entree];
        io_list[3]=gamma[marque_entree];
        }else {
        io_list[1]=p;
        io_list[3]=g;
        }
       }else {
        if((i-(int)(pow(2.0,marque_entree+1)-
1))%(int)(pow(2.0,marque_entree+2))==0) {
        io_list1[0]=pi[marque_entree];
        p_i=allocate_net(base,1);
        io_list1[1]=p_i;

instans_of_component(base,INVERTER,io_list1);
        io_list1[0]=gamma[marque_entree];
        g_i=allocate_net(base,1);
        io_list1[1]=g_i;

instans_of_component(base,INVERTER,io_list1);
        io_list[1]=p_i;
        io_list[3]=g_i;
        }else {
        io_list1[0]=p;
        p_i=allocate_net(base,1);
        io_list1[1]=p_i;

instans_of_component(base,INVERTER,io_list1);
        io_list1[0]=g;
        g_i=allocate_net(base,1);
        io_list1[1]=g_i;

instans_of_component(base,INVERTER,io_list1);
        io_list[1]=p_i;
        io_list[3]=g_i;
        }
       }
      }else { /*flag_entree = 0 */
       if(j%2==1) {
        io_list1[0]=get_port(base,1);
        io_list1[0].size=1;
        io_list1[0].number+=i;
        p_i=allocate_net(base,1);
        io_list1[1]=p_i;

instans_of_component(base,INVERTER,io_list1);
        io_list1[0]=get_port(base,2);
        io_list1[0].size=1;
        io_list1[0].number+=i;
        g_i=allocate_net(base,1);
```

```
        io_list1[1]=g_i;

instans_of_component(base,INVERTER,io_list1);
        io_list[1]=p_i;
        io_list[3]=g_i;
       }else {
        io_list[1]=get_port(base,1);
        io_list[1].size=1;
        io_list[1].number+=i;
        io_list[3]=get_port(base,2);
        io_list[3].size=1;
        io_list[3].number+=i;
       }
     }

   }

   /* les sorties des cellules d'une colonne */
   if(flag_sortie==0) {
    if(j%2==0) {
      io_list[4]=get_port(base,3);
      io_list[4].size=1;
      io_list[4].number+=i;
      io_list[5]=get_port(base,4);
      io_list[5].size=1;
      io_list[5].number+=i;

instans_of_component(base,CELLULE_BRENT_KU
NG1,io_list);
     }else {
      if((i-(int)(pow(2.0,j+1)-
1))%(int)(pow(2.0,j+2))==0) {
        pi[j]=allocate_net(base,1);
        gamma[j]=allocate_net(base,1);
        io_list[4]=pi[j];
        io_list[5]=gamma[j];

instans_of_component(base,CELLULE_BRENT_KU
NG2,io_list);
        io_list1[0]=pi[j];
        io_list1[1]=get_port(base,3);
        io_list1[1].size=1;
        io_list1[1].number+=i;

instans_of_component(base,INVERTER,io_list1);
        io_list1[0]=gamma[j];
        io_list1[1]=get_port(base,4);
        io_list1[1].size=1;
        io_list1[1].number+=i;

instans_of_component(base,INVERTER,io_list1);
      }else {
        p=allocate_net(base,1);
        g=allocate_net(base,1);
        io_list[4]=p;
        io_list[5]=g;

instans_of_component(base,CELLULE_BRENT_KU
NG2,io_list);
        io_list1[0]=p;
        io_list1[1]=get_port(base,3);
        io_list1[1].size=1;
        io_list1[1].number+=i;

instans_of_component(base,INVERTER,io_list1);
        io_list1[0]=g;
        io_list1[1]=get_port(base,4);
        io_list1[1].size=1;
        io_list1[1].number+=i;

instans_of_component(base,INVERTER,io_list1);
      }
}
   }else { /* flag_sortie==1 */
      if((i-(int)(pow(2.0,j+1)-
1))%(int)(pow(2.0,j+2))==0) {
        pi[j]=allocate_net(base,1);
        gamma[j]=allocate_net(base,1);
        io_list[4]=pi[j];
        io_list[5]=gamma[j];
        if(j%2==0)

instans_of_component(base,CELLULE_BRENT_KU
NG1,io_list);
        else

instans_of_component(base,CELLULE_BRENT_KU
NG2,io_list);
      }else {
        p=allocate_net(base,1);
        g=allocate_net(base,1);
        io_list[4]=p;
        io_list[5]=g;
        if(j%2==0)

instans_of_component(base,CELLULE_BRENT_KU
NG1,io_list);
        else

instans_of_component(base,CELLULE_BRENT_KU
NG2,io_list);
      }
    }

  }

 }
return base;
}

/*** Adder Fault-Secure using groups of parity ***/
component_lib_type
*adder_fs_with_groups_of_parity(char *name,
unsigned width)
{
 component_lib_type *base, *temp, *temp1;
 net_tuple
io_list[MAX_NETS],io_list1[MAX_NETS];
 net_tuple
BUS1,BUS2,BUS3,p_small_bus,g_small_bus,p_big_
bus,g_big_bus,carry_bus;
 unsigned name_id,fa_type,base_value;
```

```
 int i,j,Na,Nb,k,which,cas,npg;

 if (name==NULL)
 {
   char temp_string[255];
   sprintf(temp_string,"Adder_%d_Bits",width);
   base=new_component(temp_string);
 }
 else
   base=new_component(name);
 base->id=temp_comp_id;
 temp_comp_id++;
 printf("\n give the number of parity groups : ");
 scanf("%d",&npg);

 build_external_port(base,"A",IN,width);
 build_external_port(base,"B",IN,width);
 build_external_port(base,"GND",IN,1);
 build_external_port(base,"SUM",OUT,width+1);
 build_external_port(base,"PA",IN,npg);
 build_external_port(base,"PB",IN,npg);
 build_external_port(base,"PS",OUT,npg);
 build_external_port(base,"F0",OUT,npg);
 build_external_port(base,"F1",OUT,npg);

    printf("\n which structure of parallel computation
you want to generate :
            1- Kogge-Stone
            2- Han-Carlson
            3- Brent-Kung
            4- Sklanski
            5- Carry Lookahead\n");
   scanf("%d",&which);

 /* 1- generation of the propagate and generate block
*/
 temp=slice_generate_propagate(NULL,width);
 name_id=temp->id;
 insert_component_in_lib(temp);
 p_small_bus=allocate_net(base,width);
 g_small_bus=allocate_net(base,width);
 io_list[0]=get_port(base,1);
 io_list[1]=get_port(base,2);
 io_list[2]=p_small_bus;
 io_list[3]=g_small_bus;
 instans_of_component(base,name_id,io_list);

 /*2- generation of the parallel computation network
block */
 switch (which)
 {
 case 1 :
temp=n_kogge_stone(NULL,width);cas=0;break;
 case 2 : temp=n_Han_Carlson(NULL,width);cas=1;
break;
 case 3 : temp=n_Brent_Kung(NULL,width);cas=1;
break;
 case 4 : temp=n_sklanski(NULL,width);cas=2;
break;
 case 5 :
temp=n_cla_with_gr(NULL,width,npg);cas=1;break;
 default : printf("\n Error : Choice not available !\n");
  break;
 }
 name_id=temp->id;
 insert_component_in_lib(temp);
 net_tuple c[npg],c_bar[npg];
 if(which!=5) {  /**** If not carry look ahead ****/
  p_big_bus=allocate_net(base,width);
  g_big_bus=allocate_net(base,width);
  io_list[0]=p_small_bus;
  io_list[1]=g_small_bus;
  io_list[2]=p_big_bus;
  io_list[3]=g_big_bus;
  instans_of_component(base,name_id,io_list);
  /* generation of carry_generator block */

temp=carry_g_with_groups_of_carries(NULL,width,
cas,npg);
  name_id=temp->id;
  insert_component_in_lib(temp);
  io_list[0]=p_big_bus;
  io_list[1]=g_big_bus;
  io_list[2]=get_port(base,3);
  for(i=0;i<npg;i++) {
    c[i]=allocate_net(base,(int)((width-1-i)/npg+1));
    c_bar[i]=allocate_net(base,(int)((width-1-
i)/npg+1));
    io_list[3+i]=c[i];
  }
 instans_of_component(base,name_id,io_list);

 }else {  /**** If carry look ahead ****/
  io_list[0]=p_small_bus;
  io_list[1]=g_small_bus;
  io_list[2]=get_port(base,3);
  for(i=0;i<npg;i++) {
    c[i]=allocate_net(base,(int)((width-1-i)/npg+1));
    c_bar[i]=allocate_net(base,(int)((width-1-
i)/npg+1));
    io_list[3+i]=c[i];
  }
 instans_of_component(base,name_id,io_list);
 }

 /* generation of the final xor block */

temp=xor_for_adders_with_groups_of_parity(NULL,
width,cas,npg);
 insert_component_in_lib(temp);
 for(i=0;i<npg;i++)
 io_list[i]=c[i];
 io_list[1+npg-1]=p_small_bus;
 io_list[2+npg-1]=get_port(base,3);
 io_list[3+npg-1]=get_port(base,4);
 instans_of_component(base,temp->id,io_list);

 /* generation of adder carries differently */
 io_list[0]=get_port(base,1);
 io_list[1]=get_port(base,2);
 io_list[2]=p_small_bus;
 for(i=0;i<npg;i++) {
```

```
    io_list[3+i]=c[i];
    io_list[3+npg+i]=c_bar[i];
   }

temp=double_addercar_gen_with_g(NULL,width,np
g);
  name_id=temp->id;
  insert_component_in_lib(temp);
  instans_of_component(base,name_id,io_list);

/* generation of the double-rail checkers for adder
carries checking */
  unsigned ll=npg;
  unsigned q[npg];
  unsigned p,mem;
  for(i=0;i<npg;i++) {
    q[i]=(width-1-i)/npg+1;
  }
  i=0;
  while((i<npg)&&(ll>0)) {
    if(q[i]!=0) {
      mem=q[i];
      temp=n_dual_rail_checker(NULL,q[i]);
      name_id=temp->id;
      insert_component_in_lib(temp);
     for(p=i;p<npg;p++) {
      if(q[p]==mem) {
      q[p]=0;
      ll--;
      io_list[0]=c[p];
      io_list[1]=c_bar[p];
      io_list[2]=get_port(base,8);
      io_list[2].size=1;
      io_list[2].number+=(p+1)%npg;
      io_list[3]=get_port(base,9);
      io_list[3].size=1;
      io_list[3].number+=(p+1)%npg;
      instans_of_component(base,name_id,io_list);
      }
     }
    }
  i++;
  }
  net_tuple wire;
  /*generation of parity bits */
  for(i=0;i<npg;i++) {
    io_list[0]=get_port(base,8);
    io_list[0].size=1;
    io_list[0].number+=i;
    io_list[1]=get_port(base,5);
    io_list[1].size=1;
    io_list[1].number+=i;
    wire=allocate_net(base,1);
    io_list[2]=wire;
    instans_of_component(base,XOR2_CELL,io_list);
    io_list[0]=wire;
    io_list[1]=get_port(base,6);
    io_list[1].size=1;
    io_list[1].number+=i;
    io_list[2]=get_port(base,7);
    io_list[2].size=1;
    io_list[2].number+=i;
    instans_of_component(base,XOR2_CELL,io_list);
  }
  return base;
}
```

Outils de CAO pour la Génération d'Opérateurs Arithmétiques Auto-

Les chemins de données sont des parties logiques essentielles dans les microprocesseurs et les microcontrôleurs. La conception de chemins de données fiables est donc un pas important vers la réalisation de circuits intégrés plus sûrs. Nous avons, d'abord, étudié des multiplieurs auto-contrôlables basés sur le code résidu. Nous avons montré qu'on peut avoir des multiplieurs sûrs en présence de fautes de type collage logique avec un surcoût très faible, notamment pour les multiplieurs de grandes tailles (de 10 à 15% pour les multiplieurs de taille 32x32). Dans la deuxième partie, nous avons généralisé des solutions auto-contrôlables existantes d'opérateurs arithmétiques basés sur la
nouvelles versions ont plusieurs bits de parité et permettent d'augmenter sensiblement la couverture de fautes transitoires. Les solutions développées sont intégrées dans un outil informatique. Cet outil donne une grande flexibilité de choix d'
et logiques auto contrôlables permettant ainsi de faciliter la tâche des concepteurs non

: Circuits auto-contrôlables, chemins de données.

CAD Tools for the Generation of Self-Checking Arithmetic Operators

With the increasing integration density, new generations of integrated circuits are becoming more and more sensitive to noise sources such like power line disturbances, electromagnetic influence and cosmic particles. Face to this challenge, self-checking techniques could provide good solutions. In this work, we show that self-checking multipliers based on residue codes can result in a very little hardware overhead, especially for large multipliers. In a second time, we generalize fault secure solutions for multipliers, adders and shift registers based on the parity code. The new versions have several parity bits in order to increase transient coverage fault. We have implemented the presented solutions in a CAD tool. This tool offers many different self-checking arithmetic and logical operators to make flexible the construction of self-checking data paths.

Key words : self-checking, data-paths.

ISBN 2-913329-70-5 (broché)
ISBN 2-913329-71-3 (électronique)